UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Proyecto Final de Carrera

Verificación automática de protocolos criptográficos de seguridad

Marzo de 2012, Valencia

Antonio González Burgueño
angonbur@inf.upv.es

Dirigido por: Dr. Santiago Escobar Román
sescobar@dsic.upv.es

1. Introducción.

El desarrollo de este proyecto final de carrera tiene como objetivo verificar diversos protocolos de seguridad existentes utilizando una herramienta avanzada y automática de verificación de protocolos. La mayoría de los protocolos a analizar tienen ataques ya conocidos y el objetivo es comprobar si la herramienta Maude-NPA es apropiada para el modelado y verificación de protocolos con distintas propiedades. La herramienta Maude-NPA es una herramienta de verificación de propiedades de secreto y autenticación en protocolos de comunicación con propiedades criptográficas que ha sido desarrollada por el profesor Santiago Escobar de la Universidad Politécnica de Valencia en colaboración con el profesor José Meseguer (Universidad de Illinois en Urbana-Champaign, EE.UU.) y la profesora Catherine Meadows (Marina de los Estados Unidos, Washington, D.C, EE.UU.).

1.1. Motivación.

La motivación principal para la realización de este proyecto ha sido la escasa información disponible en cuanto a la especificación de protocolos de diferentes características en Maude-NPA. Los diferentes protocolos y sus especificaciones en Maude-NPA los podemos encontrar en [22].

Para esta tarea hemos elegido diferentes tipos de protocolos. Desde protocolos que utilizan un servidor de confianza como los desarrollados en la Sección 5, pasando por protocolos con autenticación repetida de clave simétrica como los desarrollados en la Sección 6, hasta protocolos con propiedades algebraicas más complejas como los desarrollados en la sección 7. Además existen protocolos que utilizan "Timestamps" pero que como Maude-NPA no puede manejarlos hemos especificado versiones alternativas que no los utilizan como los desarrollados en las secciones 5.1, 5.4 y 6.3

También hemos probado la herramienta con los diferentes protocolos y las diferentes propiedades de los mismos para comprobar la robustez y la eficacia de la herramienta. Hemos utilizado para obtener los resultados de este proyecto la herramienta en versión gráfica de Maude-NPA disponible en [23].

2. Maude-NPA.

En esta sección se presenta el Maude-NPA (Maude-NRL Protocol Analyzer). Para más información, consúltese [6]. En la Sección 2.1 se describe cómo especifica un protocolo en Maude-NPA y en la Sección 2.2 se explica cómo analizar un protocolo ya especificado.

2.1. Especificación de protocolos en Maude-NPA.

El propósito de esta sección es describir paso a paso cómo especificar un protocolo y todos los elementos relevantes del Maude-NPA. Para que el lector pueda seguir esta sección con más facilidad, se utiliza un protocolo a modo de ejemplo que ilustra todo el proceso de especificación.

2.1.1. Plantillas y organización de ficheros para la especificación de protocolos.

La especificación de un protocolo se define en un único fichero (por ejemplo foo.maude), y consiste en tres módulos Maude con un formato y un nombre de módulo ya fijados. En el primer módulo se especifica la sintaxis del protocolo, la cual consiste en tipos, subtipos y operadores. Para esto se sigue la sintaxis y las restricciones propias del lenguaje Maude [7]. El segundo módulo especifica las propiedades algebraicas de los operadores de acuerdo a la sintaxis de Maude.

El tercer módulo especifica el comportamiento propio del protocolo utilizando la notación teórica de los strands [8]. Este tercer módulo incluye los strands del intruso (llamados strands Dolev-Yao) y strands legítimos que describen el comportamiento de los participantes. En este módulo también se especifican los estados de ataque, los cuales describen el comportamiento que se desea comprobar que no se produzcan, pero se explicarán con más detalle en la Sección 2.2.

A continuación se proporciona una plantilla para cualquier especificación de protocolos en Maude-NPA. Las líneas que comienza con tres o más guiones (es decir, ---) o tres o más asteriscos (esto es, ***) son comentarios que son ignorados por Maude. Además, la sintaxis de Maude es casi auto explicativa [7]. En general, cada elemento sintáctico (por ejemplo, un tipo, una operación, una ecuación, una regla) se declara con una palabra clave obvia: sort, op, eq, rl, etc., terminada con un espacio en blanco y un punto.

```
fmod PROTOCOL-EXAMPLE-SYMBOLS is
protecting DEFINITION-PROTOCOL-RULES .
------------------------------------------------------------
--- Overwrite this module with the syntax of your protocol.
--- Notes:
```

```
--- * Sorts Msg and Fresh are special and imported
--- * Every sort must be a subsort of Msg
--- * No sort can be a supersort of Msg
--- * Variables of sort Fresh are really fresh 14

---
and no substitution is allowed on them
--- * Sorts Msg and Public cannot be empty
------------------------------------------------------------
endfm
fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
protecting PROTOCOL-EXAMPLE-SYMBOLS .
------------------------------------------------------------------
--- Overwrite this module with the algebraic properties
--- of your protocol.
--- * Use only equations of the form (eq Lhs = Rhs [nonexec] .)
--- * Maude attribute owise cannot be used
--- * There is no order of application between equations
------------------------------------------------------------
endfm
fmod PROTOCOL-SPECIFICATION is
protecting PROTOCOL-EXAMPLE-SYMBOLS .
protecting DEFINITION-PROTOCOL-RULES .
protecting DEFINITION-CONSTRAINTS-INPUT .
------------------------------------------------------------
--- Overwrite this module with the strands
--- of your protocol and the attack states
------------------------------------------------------------
eq STRANDS-DOLEVYAO =
--- Add Dolev-Yao intruder strands here. Strands are
--- properly renamed.
[nonexec] .
eq STRANDS-PROTOCOL =
--- Add protocol strands here. Strands are properly renamed.
[nonexec] .
eq ATTACK-STATE(0) =
--- Add attack state here
--- More than one attack state can be specified, but each
--- must be identified by a number (e.g. ATTACK-STATE(1) = ...
--- ATTACK-STATE(2) = ... etc.)
[nonexec] .
endfm
--- THE FOLLOWING COMMAND HAS TO BE THE LAST ACTION !!!!
select MAUDE-NPA .
```

En los siguientes apartados se explica con más detalle cómo se especifica cada uno de estos módulos.

2.1.2. Especificación de la sintaxis del protocolo.

La sintaxis de un protocolo se especifica en el módulo PROTOCOL-EXAMPLE-SYMBOLS.

Nótese que, puesto que se esta usando Maude también como lenguaje de especificación, cada declaración debe terminar con un espacio en blanco y un punto.

Primero se deben especificar los tipos (sorts). En general, los tipos se usan para especificar distintos tipos de datos, los cuáles se usan para diferentes propósitos. Existe un tipo especial llamado Msg que representa los mensajes en un protocolo. Si únicamente se pudieran utilizar claves para el cifrado, se

debería tener un tipo Key y especificar que un operador de cifrado e sólo pueden tener un término del tipo Key como primer argumento, lo cual se puede especificar en Maude de la siguiente manera: op e : Key Msg → Msg .

Los tipos también pueden ser subtipos (subsorts) de otros tipos. Los subtipos permiten obtener una distinción más refinada de los datos dentro de un tipo concreto. Por ejemplo, se podrá tener un tipo MasterKey que fuera un subtipo de Key. Estas dos relaciones relevantes entre subtipos se especifican en Maude tal y como se muestra a continuación:

```
subsort MasterKey < Key .
subsorts PublicKey PrivateKey < Key .
```

La mayoría de los tipos son definidos por el usuario. Sin embargo, existen varios tipos especiales que son importados automáticamente por cualquier definición de un protocolo en Maude-NPA a través del módulo DEFINITION-PROTOCOL-RULES. El usuario debe estar seguro de que se satisfacen las siguientes restricciones para los tipos y subtipos especificados en PROTOCOL-EXAMPLE-SYMBOLS:

- Msg. Cualquier tipo especificado en un protocolo debe estar conectado a tipo Msg, es decir, para cualquier tipo S dado por el usuario y para cualquier término t de tipo S debe existir un término ti en Msg tal que t sea un subtérmino de t_i. Cualquier subtipo de Msg se conecta directamente con Msg. Ningún tipo definido por el usuario puede ser un supertipo de Msg. El tipo Msg no puede ser vacío, es decir, que es necesario definir los suficientes símbolos de función y constantes de forma que exista al menos un término totalmente instanciado (es decir, un término sin variables) de tipo Msg.
- Fresh. El tipo Fresh se utiliza para identificar términos que deben ser únicos. Ningún tipo puede ser un subtipo de Fresh. Normalmente se usa en las especificaciones de protocolos como un argumento de algún dato que debe ser único, tal como un nonce o una clave de sesión, es decir, "n(A,r)" o "k(A,B,r)" donde r es una variable de tipo Fresh. No es necesario definir símbolos de tipo Fresh, es decir, el tipo Fresh puede estar vacío, dado que sólo se permitirán variables de ese tipo.
- Public. El tipo Public se utiliza para identificar términos que están públicamente disponibles y, por tanto, se asume que son conocidos por el intruso. Ningún tipo puede ser un supertipo de Public. Este tipo no puede estar vacío.

Para ilustrar la definición de tipos, se utilizará el protocolo de Clave Pública Needham-Schroeder (NSPK) como ejemplo. Este protocolo utiliza criptografía de clave pública y los participantes intercambian información cifrada consistente en nombres y nonces. A continuación se muestra la especificación informal de NSPK en la notación (Alice-Bob) A → B : M , donde el participante Alice (A) envía el mensaje M al participante Bob (B).

1. A → B : $\{A, N_A\}_B$
2. B → A : $\{N_A, N_B\}_A$
3. A → B : $\{N_B\}_B$

En este protocolo N_A y N_B son nonces generados por los respectivos participantes $\{M\}_A$ indica que el mensaje M ha sido cifrado con la clave pública del participante A.

Por consiguiente, se deberá definir los tipos para distinguir nombres, claves, nonces y datos cifrados. Esto se especifica tal y como se muestra a continuación:

```
--- Sort Information
sorts Name Nonce Enc .
subsort Name Nonce Enc < Msg .
subsort Name < Public .
```

Los tipos Nonce y Enc no son estrictamente necesarios, pero hacen que la búsqueda sea más eficiente, puesto que Maude-NPA no intentará unificar términos con tipos incompatibles. Por ejemplo, si en esta especificación un participante está esperando un término de tipo Enc, él o ella no aceptarán un término de tipo Nonce; técnicamente porque Nonce no se ha declarado como un subtipo de Enc. Si se están buscando ataques por confusión de tipos, no se desearía incluir estos tipos y, en su lugar, se declararía todo de tipo Msg o Name.

A continuación, se pueden especifican operadores necesarios en NSPK. Estos son pk y sk, para el cifrado público y privado, el operador n para nonces, constantes designadas para los nombres de los participantes y la concatenación usando el operador infijo "_;_".

Comencemos con los operadores del cifrado público y privado, los cuales se especifican en Maude tal y como se muestra a continuación:

```
--- Encoding operators for public/private encryption
op pk : Name Msg -> Enc [frozen] .
op sk : Name Msg -> Enc [frozen] .
```

El atributo frozen es técnicamente necesario para indicar a Maude que no intente aplicar reescrituras en los argumentos de estos símbolos.

El atributo frozen debe estar incluido en todas las declaraciones de operadores en las especificaciones en Maude-NPA, exceptuando las constantes. El uso del tipo Name como un argumento de cifrado de claves público pueden parecer extraño al principio, pero se utiliza porque implícitamente se está asociando una clave pública con un nombre cuando se aplica el operador pk y una clave privada con un nombre cuando se aplica el operador sk. Sin embargo, se podría haber elegido una sintaxis diferente para especificar claves explícitamente para el cifrado público o privado.

El siguiente paso consiste en especificar algunos nombres de participantes. Para NSPK se tienen tres constantes de tipo Name, a (para "Alice"), b (para "Bob"), y i (para el "Intruso").

```
--- Principals
op a : -> Name . --- Alice
op b : -> Name . --- Bob
op i : -> Name . --- Intruder
```

Sin embargo, estos no son todos los nombres de participantes posibles. Dado que Maude-NPA es una herramienta con un número ilimitado de sesiones, el número de participantes posibles es ilimitado. Esto se consigue utilizando variables (es decir, variables de tipo Name en esta especificación de NSPK) en lugar de constantes. Sin embargo, sería posible que se necesitara especificar nombres de participantes constantes en un estado objetivo. Por ejemplo, si se tiene un iniciador y un respondedor y no se está interesado en el caso en el cual el iniciador y el respondedor son el mismo, se puede evitar esto especificando los nombres de ambos como constantes diferentes. Además, se podría desear identificar el nombre del intruso con una constante, de forma que se cubra el caso en el cual los participantes se están comunicando directamente con el intruso.

En este punto, se necesitan dos operadores más, uno para los nonces y otro para la concatenación. El operador para los nonces se especifica tal y como se muestra a continuación:

```
--- Nonce operator
op n : Name Fresh -> Nonce [frozen] .
```

Nótese que el operador para los nonces tiene un argumento de tipo Fresh para asegurar la unicidad. El argumento de tipo Name no es estrictamente necesario pero proporciona un modo conveniente de identificar que nonces han sido generados por cada participante. Esto hace que las búsquedas sean más eficientes, puesto que permiten seguir la pista del creador de un nonce a través de una búsqueda de un ataque.

Finalmente, se especifica el operador de concatenación de un mensaje. En Maude-NPA se especifica la concatenación mediante un operador infijo "_;_" definido del siguiente modo:

```
--- Concatenation operator
op _;_ : Msg Msg → Msg [gather (e E) frozen] .
```

El atributo Maude "gather (e E)" del operador indica que el símbolo "_;_" tiene que se analizado sintácticamente como asociado por la derecha, mientras que "gather (E e)" indica asociación por la derecha. Nótese que esta información recogida se especifica únicamente con propósito para el análisis sintáctico: no se especifica una propiedad asociativa, aunque la asociatividad de un operador se pueda expresar en Maude bien como una "regla ecuacional" explícita o como un "axioma ecuacional" como se explica en la Sección 2.1.3. Por tanto, "_;_" en este ejemplo de NSPK es tan sólo un

símbolo de función libre, el cual, cuando no se le ha añadido ningún paréntesis, es analizado siguiendo un modo asociativo por la izquierda.

2.1.3. Propiedades algebraicas.

Existen dos tipos de propiedades algebraicas: (i) axiomas ecuacionales, tales como conmutatividad o asociatividad-conmutatividad, llamados axiomas, y (ii) reglas ecuacionales, llamadas ecuaciones. Las ecuaciones se especifican en el módulo PROTOCOL-EXAMPLE-ALGEBRAIC, mientras que los axiomas se especifican en las declaraciones de operadores en el módulo PROTOCOL-EXAMPLE-SYMBOLS tal y como se muestra a continuación.

Una ecuación se orienta en una regla de reescritura en la cual la parte izquierda de la ecuación se reduce a la parte derecha. En las ecuaciones escritas, es necesario especificar las variables involucradas, así como su tipo. Las variables pueden especificarse globalmente para un módulo, como por ejemplo, "var Z : Msg .", o localmente dentro de la expresión que la utiliza, como por ejemplo, una variable A de tipo Name en "pk(A:Name,Z)". Se pueden especificar juntas distintas variables del mismo tipo como "vars X Y Z1 Z2 : Msg .". En NSPK se utilizan dos ecuaciones para especificar la relación entre el cifrado de clave pública y privada, tal y como se muestra a continuación:

```
var Z : Msg .
var A : Name .
--- Encryption/Decryption Cancellation
eq pk(A,sk(A,Z)) = Z [nonexec metadata "variant"]  .
eq sk(A,pk(A,Z)) = Z [nonexec metadata "variant"]  .
```

El atributo nonexec es técnicamente necesario para indicar a Maude que no use una ecuación o una regla dentro de su ejecución estándar, dado que únicamente sería utilizado en el nivel de Maude-NPA en vez de en el nivel de Maude. Se debe incluir este atributo en todas las declaraciones de ecuaciones del protocolo definidas por el usuario.

Además el atributo de Maude-NPA "owise" (es decir, otherwise) no puede ser usado en ecuaciones puesto que no se asume ningún orden de aplicación para las ecuaciones algebraicas.

Dado que Maude-NPA utiliza algoritmos de unificación empotrados [9] para el caso en el que tengan o bien no tengan axiomas, o axiomas ecuacionales conmutativos (C) o asociativo-conmutativos (AC), estos se especifican no como ecuaciones estándar sino como axiomas en la sección declaración de operadores. Por ejemplo, supóngase que se desea especificar el or exclusivo. Se puede especificar un operador infijo asociativo y conmutativo "<+>" en el módulo PROTOCOL-EXAMPLE-SYMBOLS tal y como se muestra a continuación:

```
--- XOR operator and equational axioms
op _<+>_ : Msg Msg -> Msg [frozen assoc comm] .
op null : -> Msg .
```

Donde se declaran los axiomas de asociatividad y conmutatividad como atributos del operador <+> con las palabras reservadas assoc y comm.

Las reglas ecuacionales para el operador <+> se especifican en el módulo PROTOCOL-EXAMPLE-ALGEBRAIC como ecuaciones declaradas con la palabra reservada eq tal y como se muestra a continuación

```
    vars X Y : Msg .
--- XOR equational rules
eq X <+> X <+> Y = Y [nonexec] [metadata "variant"]  .
eq X <+> X = null [nonexec] [metadata "variant"].
eq X <+> null = X [nonexec] [metadata "variant"]  .
```

2.1.4. Propiedades del operador Homomorfismo.

Maude-NPA proporciona tres tipos diferentes de propiedades algebraicas:

- Cuando los axiomas son sólo propiedades algebraicas, es el caso de cuando los símbolos solo tienen las propiedades algebraicas conmutativa, asociativa-conmutativa, o asociativa-conmutativa-identidad.

- Cuando las ecuaciones variables (con o sin axiomas) describen las propiedades algebraicas.

- Cuando las propiedades algebraicas no se ajustan a los dos casos anteriores, dedicados algoritmos de unificación para teorías ecuacionales que pueden diseñarse e implementarse en Maude-NPA. En este caso, algunas ecuaciones dedicadas están asociadas para la teoría ecuacional para que Maude-NPA pueda identificar cuando una teoría ecuacional pueda ser utilizada para la especificación de un protocolo y para invocar los algoritmos de unificación dedicada de la manera adecuada. El desarrollo de este tipo de algoritmos se ha hecho por dos propósitos.

    1. Permitir la unificación para las teorías ecuacionales que no es soportada por la combinación de axiomas y ecuaciones variables, y,

    2. Para mejorar la eficiencia sobre las teorías comúnmente utilizadas (e.g. or exclusiva o grupos abelianos).

Actualmente se ha integrado un algoritmo de unificación para la encriptación siendo homomórfico sobre la concatenación, i.e., satisfaciendo la siguiente propiedad algebraica genérica:

$$e(X;Y,Key) = e(X,Key);e(Y,Key)$$

Sin embargo, los símbolos actualmente elegidos, e y _;_ son definibles por el usuario. Por ejemplo, supongamos que queremos especificar una versión del protocolo de Needham-Schroeder-Lowe (NSL), que es una versión

arreglada para evitar fallos del protocolo de clave pública de Needham-Schroeder (NSPK) hecha por Lowe, pero incluyendo la propiedad algebraica en la que la encriptación es homomórfica sobre concatenación. Entonces, la ecuación dedicada añadida al modulo "PROTOCOL-EXAMPLE-ALGEBRAIC" es como sigue:

```
"eq pk(X:Msg ; Y:Msg, K:Key) = pk(X:Msg, K:Key) ; pk(Y:Msg, K:Key)
[nonexec label homomorphism metadata "builtin-unify"] ."
```

Similar a la inclusión de ecuaciones variables, estas ecuaciones dedicadas contienen los atributos "nonexec" y "metadata". Sin embargo, se han añadido dos nuevas características sintácticas para que Maude-NPA pueda identificar la adecuada ecuación dedicada:

1. La etiqueta "homomorphism" se utiliza explícitamente para identificar esta ecuación. Esto ayudaría en el futuro cuando diferentes algoritmos dedicados se puedan combinar.

2. La etiqueta "metadata "builtin-unify"" se utiliza para separar ecuaciones dedicadas de las ecuaciones variables. Esto ayudará en el futuro cuando las ecuaciones variables y dedicadas puedan combinarse.

Debemos fijarnos que esta propiedad ecuacional fuerza a que la clave sea el segundo argumento de la encriptación. También, solo un símbolo de la encriptación puede ser homomórfico sobre la concatenación, es decir, sólo una ecuación dedicada con la etiqueta "homomorphism" está permitida.

Además, actualmente, ninguno de los otros símbolos del protocolo pueden tener las propiedades algebraicas conmutativa, asociativa-conmutativa, o asociativa-conmutativa-identidad.

2.1.5. Especificación de protocolos: Strands del intruso.

El propio protocolo y las habilidades del intruso se especifican en el módulo PROTOCOL-SPECIFICATION. Ambos se especifican utilizando strands. Un strand, definido por primera vez en [8], es una secuencia de mensajes positivos y negativos3 describiendo un participante ejecutando un protocolo o las acciones llevadas a cabo por el intruso, es decir, el strand para Alice en NSPK es:

$$[\ pk(KB, A; NA)^{+}, pk(KA, NA; Z)^{-}, pk(KB, Z)^{+}\ ]$$

donde un mensaje positivo implica un envío y un mensaje negativo implica la recepción.

Sin embargo, en Maude-NPA cada strand se divide en la parte del pasado y la parte del futuro, mediante una línea vertical. Esto significa que los mensajes a la izquierda de la línea vertical fueron enviados o recibidos en el

pasado, mientras que los mensajes a la derecha de la línea serán enviados o recibidos en el futuro. Además, se sigue la pista de todas las variables de tipo Fresh generadas por un strand en concreto. Es decir, todas las variables r1… rj de tipo Fresh se hacen explícitas justo antes del strand, tal y como se muestra a continuación:

$$:: r_1, \ldots, r_j :: [\ m_1{}^{\pm},\ \ldots,\ m_i{}^{\pm} \mid m_{i+1}{}^{\pm},\ \ldots,\ m_k{}^{\pm}\ ]$$

Nótese que existe alguna diferencia entre las variables de tipo Fresh generadas en un strand y aquellas que aparecen en un strand. En la especificación de un rol deben coincidir, en el sentido de que todas las variables que aparecen en la descripción de un participante son generadas durante la ejecución (una instancia) de ese strand. Sin embargo, en un estado del protocolo dado, un strand puede contener muchas más variables de tipo Fresh que aquellas especificadas a la izquierda del strand, debido al intercambio de mensajes entre los participantes.

En primer lugar se especifican todas las variables que se usan en este módulo, junto con los tipos de estas variables. En el ejemplo de NSPK éstas son:

```
vars X Y Z : Msg .
vars r r' : Fresh .
vars A B : Name .
vars N N1 N2 : Nonce .
```

Una vez se han definido las variables, el siguiente paso consiste en especificar las acciones del intruso, o reglas Dolev-Yao [10]. Éstas especifican las operaciones que el intruso puede realizar. Cada una de estas acciones puede ser especificada mediante un strand del intruso que conste de una secuencia de mensajes negativos seguidos por un único mensaje positivo. Si el intruso puede encontrar, de forma no determinista, más de un término como resultado de llevar a cabo una operación (como en la desconcatenación), se especifica cada una de estas salidas mediante strands separados. Para el protocolo NSPK, tenemos cuatro operaciones: cifrado con una clave pública (pk), descifrado con una clave privada (sk), concatenación (_;_) y desconcatenación.

El cifrado con una clave pública se especifica tal y como se muestra a continuación.

Nótese que se utiliza el nombre del participante para denotar la clave. Esta es la razón por la que los nombres son de tipo Name. El intruso puede cifrar cualquier mensaje utilizando cualquier clave pública.

```
:: nil:: [ nil  |  -(X), +(pk(A,X)), nil ]
```

El cifrado con una clave privada es un poco diferente. El intruso sólo puede aplicar el operador sk utilizando su propia identidad. Por tanto, se especifica la regla del modo siguiente, asumiendo que i denota el nombre del intruso:

```
:: nil:: [ nil | -(X), +(sk(i,X)), nil ]
```

La concatenación y desconcatenación se obtienen directamente. Si el intruso sabe X e Y , entonces puede componer el mensaje X ; Y . Si él sabe X ; Y entonces puede averiguar X e Y . Dado que cada strand del intruso debería tener como mucho un mensaje positivo, es necesario utilizar tres strands para especificar estas acciones:

```
:: nil :: [ nil | -(X), -(Y), +(X ; Y), nil ]
:: nil :: [ nil | -(X ; Y), +(X), nil ]
:: nil :: [ nil | -(X ; Y), +(Y), nil ]
```

A continuación se muestra la especificación Dolev-Yao final. Nótese que Maude-NPA requiere el uso del símbolo constante STRANDS-DOLEVYAO como el repositorio de todos los strands Dolev-Yao, y utiliza el símbolo asociativo y conmutativo _&_ como el operador de unión para formar conjuntos de strands. Además, Maude-NPA considera que las variables no se comparten entre strands y que, por tanto, se renombraron apropiadamente cuando sea necesario.

```
eq STRANDS-DOLEVYAO
= :: nil :: [ nil | -(X), -(Y), +(X ; Y), nil ] &
:: nil :: [ nil | -(X ; Y), +(X), nil ] &
:: nil :: [ nil | -(X ; Y), +(Y), nil ] &
:: nil :: [ nil | -(X), +(sk(i,X)), nil ] &
:: nil :: [ nil | -(X), +(pk(Ke,X)), nil ]
[nonexec] .
```

Toda operación que pueda ser realizada por el intruso y todo término que es inicialmente conocido por el intruso debería tener un strand del intruso correspondiente.

Por cada operación utilizada en el protocolo se debería considerar si el intruso puede realizarla o no y especificar el strand del intruso correspondiente que describa las condiciones bajo las cuales el intruso puede realizarla.

Por ejemplo, supóngase que se requiere el uso del or exclusivo. Si se asume que el intruso puede realizar esta operación entre dos términos conocidos por él, se debería representar esto con el siguiente strand:

```
:: nil :: [ nil | -(X), -(Y), +(X <+> Y), nil ]
```

2.1.6. Especificación de protocolos: Strands del protocolo.

En la sección de las reglas del protocolo de una especificación se definen los mensajes que son enviados y recibidos por los participantes legítimos. Se especificará un strand por cada rol. Sin embargo, dado que el análisis de Maude-NPA soporta un número arbitrario de sesiones, cada strand puede estar instanciado un número arbitrario de veces.

En la especificación de los strands de un protocolo es importante recordar especificarlos desde el punto de vista del participante que ejecuta el rol. Por ejemplo, en NSPK el iniciador Alice comienza enviando su nombre y un nonce cifrado con la clave pública de Bob. Después, ella recibe algo cifrado con su clave pública, pero todo lo que ella puede decir es que es lo que recibe es su nonce concatenado con otro nonce. Después ella cifra ese otro nonce bajo la clave pública de Bob y lo envía.

La construcción del nonce de Alice se representa explícitamente como n(A,r), donde r es una variable de tipo Fresh que, por tanto, aparecerá en la cabecera del strand de Alice, y el nonce extra que ella recibe se representa con la variable N de tipo Nonce. A continuación se muestra el strand completo para el rol iniciador:

```
:: r ::
[ nil | +(pk(B,A ; n(A,r))),
        -(pk(A,n(A,r) ; N)),
        +(pk(B, N)), nil ]
```

En el strand del respondedor, los signos de los mensajes son los opuestos. Además, los mensajes se representan de modo diferente, Bob comienza recibiendo un nombre y un nonce cifrado con su clave. El crea su propio nonce, le concatena al final el nonce recibido, cifra todo esto con la clave perteneciente al nombre y lo envía. Después recibe su nonce cifrado con su propia clave. Esto se especifica tal y como se muestra a continuación:

```
:: r' ::
[ nil | -(pk(B,A ; N)),
        +(pk(A, N ; n(B,r'))),
        -(pk(B,n(B,r'))), nil ]
```

La especificación completa de STRANDS-PROTOCOL para NSPK se presenta a continuación:

```
eq STRANDS-PROTOCOL =
:: r ::
[nil | +(pk(B,A ; n(A,r))),
       -(pk(A,n(A,r) ; N)),
       +(pk(B, N)), nil]
&
:: r ':
[nil | -(pk(B,A ; N)),
      +(pk(A, N ; n(B,r'))),
       -(pk(B,n(B,r'))), nil]
[nonexec] .
```

Para terminar, es necesario recalcar que, si B recibió un mensaje Z cifrado con una clave que él no conoce, entonces él no sería capaz de verificar que recibió pk(A,Z) porque no puede descifrar el mensaje. En ese caso, lo mejor que podría decirse es que A recibió un término Y de tipo Msg, siendo Msg el tipo de datos más general que se puede utilizar.

La especificación completa del protocolo de clave pública Needham-Schroeder puede encontrarse en [11].

2.2. Análisis del protocolo.

En esta sección se describe cómo analizar un protocolo en la práctica. Primero se explica la apariencia de un estado de un protocolo y como se especifica un estado de ataque en el protocolo. Después, se explica cómo se lleva a cabo realmente el análisis del protocolo.

2.2.1. Estados del protocolo.

En Maude-NPA, cada estado asociado a la ejecución del protocolo (es decir, una búsqueda hacia atrás) se representa por un término con cuatro componentes diferentes separados por el símbolo | | en el siguiente orden:

(1) El conjunto de strands actuales,
(2) el conocimiento actual del intruso,
(3) la secuencia de mensajes encontrada hasta el momento en la ejecución hacia atrás, y
(4) otra información adicional.

Strands | | Intruder Knowledge | | Message Sequence | | Auxiliary Data

El primer componente, indica en particular cómo de avanzado está cada strand en el proceso de ejecución (por la posición de la barra vertical). El segundo componente contiene mensajes que el intruso ya sabe (símbolo _inI) y los mensajes que el intruso actualmente no conoce (símbolo _!inI) pero que aprenderá en el futuro. Nótese que el conjunto de strands y el conocimiento del intruso crecen conforme avanza la búsqueda hacia atrás de modo siguiente:

- Por propagación de las sustituciones computadas por la unificación módulo la teoría ecuacional,
- introduciendo más strands del protocolo o del intruso,
- introduciendo más conocimiento positivo del intruso (es decir, M inI), y
- transformando conocimiento positivo en conocimiento negativo debido a la ejecución hacia atrás (es decir M inI → M !inI).

El tercer componente, la secuencia de mensajes, es nil para cualquier estado de ataque al principio de la búsqueda hacia atrás y, al llegar a un estado inicial, recoge la secuencia real de mensajes intercambiadas hasta el momento en la búsqueda hacia atrás desde el estado de ataque. Esta secuencia crece conforme la búsqueda hacia atrás continúa y algunas variables puede estar instancias en la búsqueda hacia atrás. Esto da una descripción completa de un ataque cuando se alcanza un estado inicial, pero el propósito de este componente es, sobretodo, aportar algo más de información al usuario y actualmente no se usa en la propia búsqueda hacia atrás, excepto para almacenar dicha información. Finalmente, el último componente contiene información sobre el espacio de búsqueda que la herramienta crea para manejar su búsqueda. No proporciona ninguna información so-

bre el ataque y simplemente es mostrado por la herramienta para ayudar en la depuración. Para más información, consúltese [6].

2.2.2. Estados iniciales.

Un estado inicial es el resultado final de un proceso de búsqueda de alcanzabilidad hacia atrás y se describe tal y como se explica a continuación:

1. en un estado inicial, todos los strands tienen la barra al principio, es decir, todos los strands son de la forma

$$:: r_1, \ldots, r_j :: [\ nil \mid {m_1}^{\pm}, \ \ldots, \ {m_k}^{\pm}\ ];$$

2. en un estado inicial, todo el conocimiento del intruso es negativo, es decir, todos los elementos son de la forma !inI.

Desde un estado inicial, no se puede ejecutar ningún otro paso de alcanzabilidad hacia atrás puesto que no existe nada que el intruso pueda aprender o desaprender (es decir, M inI → M !inI) debido a los dos hechos siguientes:

1. Puesto que el conocimiento del intruso ya es negativo, !inI , no existe ningún mensaje positivo que el intruso pueda dejar de conocer y,
2. no se podrá producir ninguna otra recepción de mensajes (no habrá mensajes negativos en un strand) que pueda introducir elementos positivos inI en el conocimiento del intruso.

2.2.3. Estados inalcanzables.

Otro concepto interesante es el de un estado inalcanzable. Un estado inalcanzable es un estado del protocolo desde el cual no se puede alcanzar un estado inicial mediante un análisis de alcanzabilidad hacia atrás. Interpretado en un modo hacia adelante, esto significa que es un estado que no puede ser alcanzado nunca desde un estado inicial mediante una ejecución hacia adelante. La detección temprana de estados inalcanzables es esencial para lograr un análisis efectivo del protocolo por lo que Maude-NPA incorpora diversas técnicas que permiten realizar esta detección. Por ejemplo, considérese el siguiente estado encontrado por Maude-NPA para el NSPK:

```
:: nil :: [nil | -(pk(i, b ; n(b, r))), +(b ; n(b, r)), nil] &
:: nil :: [nil | -(n(b, r)), +(pk(b, n(b, r))), nil] &
:: nil :: [nil | -(b ; n(b, r)), +(n(b, r)), nil] &
:: r :: [nil | +(pk(i, b ; n(b, r))), nil] &
:: r :: [nil, -(pk(b, a ; N)), +(pk(a, N ; n(b, r)))
          | -(pk(b, n(b, r))), nil])
| |
pk(b, n(b, r)) !inI, pk(i, b ; n(b, r)) !inI, n(b, r) !inI, (b ; n(b, r)) !inI
| |
+(pk(i, b ; n(b, r))), -(pk(i, b ; n(b, r))), +(b ; n(b, r)),
-(b ; n(b, r)), +(n(b, r)), -(n(b, r)), +(pk(b, n(b, r))),
-(pk(b, n(b, r)))
| |
nil
```

Este estado es inalcanzable, puesto que existen dos strands que están generando la misma variable fresca r y esto es imposible porque la información fresca generada por cada strand debe ser única.

2.2.4. Estados de ataque.

Los estados de ataque describen no sólo un ataque en concreto sino también patrones de ataque (o si se prefiere situaciones de ataque), los cuales se especifican simbólicamente como términos (con variables) cuyas instancias son los estados de ataque finales que se están buscando. Dado un patrón de ataque, Maude-NPA o bien trata de encontrar una instancia del ataque o bien trata de probar que no existe ninguna instancia de ese patrón de ataque posible. Se puede especificar más de un estado de ataque. Por tanto, cada estado de ataque se identifica con un número natural.

Cuando se especifique un estado de ataque, el usuario debería especificar sólo los dos primeros componentes del estado de ataque: (i) un conjunto de los strands que se espera que aparezcan en el ataque y (ii) algún conocimiento positivo del intruso. Los otros dos componentes del estado deberían tener el símbolo vacío nil. Nótese que el estado de ataque es, es realidad, un término con variables pero en el cual el usuario no tiene que proporcionar las variables que denotan "los strands restantes", el "conocimiento del intruso restante" y las dos variables para los dos últimos componentes del estado.

Estas variables son insertadas simbólicamente por la herramienta.

Para NSPK, el ataque estándar se representa tal y como se muestra a continuación:

```
eq ATTACK-STATE(0) =
:: r ::
[ nil, -(pk(b,a ; N)), +(pk(a, N ; n(b,r))), -(pk(b,n(b,r))) | nil ]
|| n(b,r) inl
|| nil
|| nil
[nonexec] .
```

Donde se requiere que el intruso haya aprendido el nonce generado por Bob. Por tanto, se ha tenido que incluir el strand de Bob en el ataque a fin de describir el nonce específico n(b,r).

En resumen, la especificación de un estado de ataque debe seguir los requisitos que se presentan a continuación:

1. Los strands de un estado de ataque deben tener la barra al final. Esto también se aplica a los strands ejecutados parcialmente, para los cuales se descartan los mensajes del futuro.
2. Si aparece más de un strand en el estado de ataque, deben estar separados por el símbolo &. Si aparece más de un término en el conocimiento del intruso, deben estar separados mediante

comas. Si no aparece ningún strand, o ningún conocimiento del intruso, entonces deberá usarse el símbolo empty en los componentes de los strands o del conocimiento del intruso, respectivamente.

3. Los elementos que pueden aparecer en el conocimiento del intruso pueden incluir no sólo términos conocidos por el intruso, sino también condiciones de desigualdad sobre términos. Los términos desconocidos por el intruso son también posibles pero no son comunes en los estados de ataque.
4. Los dos últimos campos de un estado de ataque deben ser siempre nil.

2.2.5. Estados de ataque con patrones de exclusión: Never Patterns.

A veces es deseable excluir ciertos patrones de ejecución que conducen a un estado de ataque. Por ejemplo, uno puede querer determinar si las propiedades de autenticación han sido violados, por ejemplo, si es posible que un strand de respuesta aparezca sin un iniciador de la comunicación correspondiente. Para ello existe un campo opcional adicional en el estado que contiene esos patrones que deseamos excluir, los "Never Patterns".

Aquí podemos ver como especificaríamos el strand del iniciador de la conversación, sin el strand de respuesta, del protocol NSPK:

```
eq ATTACK-STATE(1) =
    :: r ::
          [ nil,
             -(pk(b,a ; N)),
            +(pk(a, N ; n(b,r))),
             -(pk(b,n(b,r))) | nil ]
          | | empty
          | | nil
          | | nil
          | | never(
          *** for authentication
          :: r' ::
          [ nil |
              +(pk(b,a ; N)),
               -(pk(a, N ; n(b,r))),
              +(pk(b,n(b,r))), nil ]
          & S:StrandSet | | K:IntruderKnowledge )
[nonexec] .
```

Podemos observar que el "never pattern" no contiene los campos extras incluidos en el patrón de ataque y solo están disponibles los strands y el conocimiento obtenido del intruso. Las variables de los strands o del conocimiento del intruso pueden aparecer en los "never pattern" como una manera de diferenciar y limitar los "never pattern".

Por ejemplo, la variable r' en el "never pattern" no es un error. En el strand del estado de ataque del iniciador del protocolo, que genera la variable r, el strand del "never pattern" es un patrón que debe de ser valido para cualquiera que responda, y éste podría generar su variable r'.

Si queremos restringir aun más el strand del que responde, de manera que no se puedan mostrar strands parciales, deberemos incluir un segundo "never pattern" que comience siempre con un "+" (así representaremos los strands incompletos).

```
eq ATTACK-STATE(1) =
        :: r ::
        [ nil,
           -(pk(b,a ; N)),
          +(pk(a, N ; n(b,r))),
           -(pk(b,n(b,r))) | nil ]
        || empty
        || nil
        || nil
        || never(
        *** for authentication
        (:: r' ::
        [ nil |
            +(pk(b,a ; N)),
             -(pk(a, N ; n(b,r))),
            +(pk(b,n(b,r))), nil ]
        & S:StrandSet || K:IntruderKnowledge)
        *** for authentication
        (:: r' ::
        [ nil |
            +(pk(b,a ; N)), nil ]
                & S:StrandSet || K:IntruderKnowledge) )
        [nonexec] .
```

Podemos observar que los nombres de las variables usados en los diferentes "never patterns" no tienen efecto, ya que cada patrón es comprobado de manera independiente. La herramienta buscará todas las rutas donde el strand del intruso es ejecutado, pero ningún strand parcial del que responde se está ejecutando.

También es posible usar los "never pattern" para especificar condiciones negativas en términos o strands. Supongamos que queremos preguntar si es posible para el que responde en el protocol NSPK ejecutar una sesión del protocolo, aparentemente con el iniciador, pero el nonce recibido no es el del iniciador (representado por "n(a,r')". Esto podríamos mostrarlo de la siguiente manera.

```
eq ATTACK-STATE(2) =
        :: r ::
        [ nil,
            -(pk(b,a ; N)),
           +(pk(a, N ; n(b,r))),
            -(pk(b,n(b,r))) | nil ]
        || empty
        || nil
        || nil
        || never(
*** for authentication (
        :: r' ::
        [ nil |
            -(pk(b,a ; n(a,r'))),
           +(pk(a, n(a,r') ; n(b,r))),
            -(pk(b,n(b,r))), nil ]
        & S:StrandSet || K:IntruderKnowledge) )
[nonexec] .
```

También es posible poner más de un "never pattern" en un espacio de búsqueda, pero entonces cada patrón debe estar contenido entre paréntesis de la siguiente manera:

```
never(
 ( ... State 1 ... )
 ( ... State 2 ... )
 )
```

Los "never pattern" también se pueden utilizar para recortar el espacio de búsqueda. Supongamos, por ejemplo, que encontramos en una búsqueda un numero de estados en los cuales el intruso encripta dos "nonces", pero éstos no nos proporcionan ninguna información útil. Podríamos reducir el espacio de búsqueda eliminando este conocimiento del intruso usando los siguientes "never pattern":

```
eq ATTACK-STATE(1) =
  :: r ::
  [ nil,
      -(pk(b,a ; N)),
     +(pk(a, N ; n(b,r))),
      -(pk(b,n(b,r))) | nil ]
  || empty
  || nil
   || nil || never(
*** cut down search for two nonces
  :: nil ::
  [ nil |
       -(N1 ; N2),
      +(pk(A, N1 ; N2)), nil ]
  & S:StrandSet || K:IntruderKnowledge )
[nonexec] .
```

Podemos remarcar que, añadiendo "never patterns" para reducir el espacio de búsqueda, distinguiéndolo de su uso para verificar propiedades de autenticación, significa que si no se encuentra un ataque no quiere decir necesariamente que el protocolo sea seguro. Simplemente significa que ningún ataque contra la propiedad especificada debe de usar ,al menos, un strand especificado en el "never pattern".

Especificaremos también algunas cosas importantes sobre los "never pattern":

1. La barra, en cualquier strand en el "never pattern" debe de ir al principio del strand.
2. Los dos primeros campos, deben de terminar en variables de tipo "Strandset" e "Intruderknowledge" respectivamente.
3. Se pueden utilizar más de un "never pattern" en los estados de ataque. Cada uno puede delimitarse por sus propios paréntesis

2.2.6. Comandos de Maude-NPA para la búsqueda de un ataque.

Los comandos "run", "summary" e "initials" son los comandos que la herramienta proporciona para realizar la búsqueda de un ataque. Estos comandos se invocan mediante su reducción en Maude, es decir, escribiendo el comando de Maude-NPA "red" seguido del comando Maude-NPA correspondiente, seguido por un espacio en blanco y un punto.

Para usarlos, se debe especificar el estado de ataque que se está buscando y el número de pasos de búsqueda hacia atrás que se desea computar. Por ejemplo, el comando Maude-NPA "run(0,10)" indica a Maude-NPA que debe construir el árbol de búsqueda hacia atrás de hasta 10 niveles de profundidad para el estado de ataque identificado con el número natural 0.

El comando de Maude-NPA "run" produce el conjunto de estados encontrados en las hojas del árbol de búsqueda hacia atrás de la profundidad especificada que ha sido generado.

Cuando el usuario no está interesado en los estados actuales del árbol de búsqueda, puede usar el comando "summary" de Maude-NPA, el cual sólo produce el número de estados encontrados en las hojas del árbol de búsqueda y cuántos de ellos son estados iniciales, es decir, soluciones del ataque. Por ejemplo, cuando se proporciona el comando de reducción en Maude con el comando Maude-NPA "summary(0,2)" para el ejemplo NSPK tal y como se muestra a continuación, la herramienta devuelve:

```
red summary(0,2) .
result Summary: States>> 4 Solutions>> 0
```

El estado inicial que representa el ataque estándar del protocolo NSPK se encuentra en siete pasos. Esto significa que si se escribe el comando "red summary(0,7)" . Maude-NPA produce la siguiente salida:

```
red summary(0,7) .
result Summary: States>> 3 Solutions>> 1
```

Una versión ligeramente diferente del comando "run", llamado "initials", produce como salida únicamente los estados iniciales, en lugar de todos los estados de las hojas del árbol de búsqueda hacia atrás. Por tanto, si se escribe en el siguiente comando:

```
red initials(0,7) .
```

Para el ejemplo de NSPK, la herramienta produce el siguiente estado inicial, lo cual significa que se ha demostrado que el estado de ataque es alcanzable y, por tanto, el protocolo no es seguro.

```
Maude> red initials(0,7) .
result IdSystem: < 1 . 5 . 2 . 7 . 1 . 4 . 3 . 1 > (
:: nil :: [nil | -(pk(i, n(b, #1:Fresh))), +(n(b, #1:Fresh)), nil] &
:: nil :: [nil | -(pk(i, a ; n(a, #0:Fresh))), +(a ; n(a, #0:Fresh)), nil] &
:: nil :: [nil | -(n(b, #1:Fresh)), +(pk(b, n(b, #1:Fresh))), nil] &
:: nil :: [nil | -(a ; n(a, #0:Fresh)), +(pk(b, a ; n(a, #0:Fresh))), nil] &
```

Maude-NPA asocia un identificador, por ejemplo "1 . 5 . 2 . 7 . 1 . 4 . 3 . 1", para cada estado generado por la herramienta. Estos identificadores son para uso interno y no se describen aquí.

```
:: #1:Fresh ::
 [nil | -(pk(b, a ; n(a, #0:Fresh))),
        +(pk(a, n(a, #0:Fresh) ; n(b, #1:Fresh))),
        -(pk(b, n(b, #1:Fresh))), nil] &
:: #0:Fresh ::
 [nil | +(pk(i, a ; n(a, #0:Fresh))),
        -(pk(a, n(a, #0:Fresh) ; n(b, #1:Fresh))),
        +(pk(i, n(b, #1:Fresh))), nil])
| |
pk(a, n(a, #0:Fresh) ; n(b, #1:Fresh)) !inI, pk(b, n(b, #1:Fresh)) !inI,
pk(b, a ; n(a, #0:Fresh)) !inI,
pk(i, n(b, #1:Fresh)) !inI,
pk(i, a ; n(a, #0:Fresh)) !inI,
n(b, #1:Fresh) !inI,
(a ; n(a, #0:Fresh)) !inI
| |
+(pk(i, a ; n(a, #0:Fresh))),
-(pk(i, a ; n(a, #0:Fresh))),
+(a ; n(a, #0:Fresh)),
-(a ; n(a, #0:Fresh)),
+(pk(b, a ; n(a, #0:Fresh))),
-(pk(b, a ; n(a, #0:Fresh))),
+(pk(a, n(a, #0:Fresh) ; n(b, #1:Fresh))),
-(pk(a, n(a, #0:Fresh) ; n(b, #1:Fresh))),
+(pk(i, n(b, #1:Fresh))),
-(pk(i, n(b, #1:Fresh))),
+(n(b, #1:Fresh)),
-(n(b, #1:Fresh)),
+(pk(b, n(b, #1:Fresh))),
-(pk(b, n(b, #1:Fresh)))
| |
nil
```

Esto se corresponde con la siguiente versión textual del ataque:

1. A → I : pk(I, A; NA)
2. I_A → B : pk(B, A; NA)
3. B → A : pk(A, NA; NB), interceptado por I;
4. I → A : pk(A, NA; NB)
5. A → I : pk(I, NB)
6. I_A → B : pk(B, NB)

3. Una sesión de ejemplo con la interfaz.

La versión actual de la IGU desarrollada para Maude-NPA permite al usuario analizar la especificación de un protocolo criptográfico, mediante la representación grafica del espacio de búsqueda generado por Maude-NPA como un árbol donde cada nodo ´representa un posible estado de ejecución del protocolo. Además, el usuario puede obtener la información textual de cada estado del árbol y, como una funcionalidad especial, se puede obtener una representación visual de los strands de un estado. A lo largo de esta sección se explicara cómo funciona la herramienta, tomando como ejemplo el famoso protocolo de clave pública Needham-Schroeder (NSPK) [12]. A continuación se recuerda la especificación informal del NSPK siguiendo la notación A → B: M donde el participante A envía el mensaje M al participante B:

$$A \rightarrow B : \{N_A, A\}_B$$
$$B \rightarrow A : \{N_A, N_B\}_A$$
$$A \rightarrow B : \{N_B\}_B$$

donde N_A y N_B son nonces generados por los respectivos participantes de la sesión (A y B) y la notación $\{M\}_C$ indica que el mensaje M ha sido encriptado con la clave pública del participante C y, por lo tanto, sólo él puede obtener el mensaje M usando su clave privada. Utilizando strands, el protocolo NSPK se describiría de la siguiente forma, donde $\{M\}_C$ se escribe simbólicamente como pke(C,M) y un nonce N_C como n(C,r):

$$:: r :: [\ pke(B, n(A, r); A)^{+}, pke(A, n(A, r); NB)^{-}, pke(B, NB)^{+}\ ]$$
$$:: r' :: [\ pke(B, NA; A)^{-}, pke(A, NA; n(B, r'))^{+}, pke(B, n(B, r'))^{-}\ ]$$

El primer paso es seleccionar el protocolo que se desea analizar. El usuario puede cargar su propio archivo con la especificación del protocolo o seleccionar uno de los ejemplos que la herramienta proporciona. Puesto que pueden definirse más de un estado de ataque en la especificación del protocolo, el usuario debe seleccionar qué estado de ataque en concreto desea analizar. Por ejemplo, en la especificación del protocolo NSPK se han definido dos estados de ataque: a0 y a1, por lo que, una vez cargado el protocolo, se mostrará al usuario la ventana mostrada en la Figura 5. Una vez el protocolo, el estado de ataque y (posiblemente) las gramáticas han sido elegidos, aparece una nueva ventana con el árbol de búsqueda inicializado como puede verse en la Figura 6. Este árbol inicial contiene un nodo llamado "nodo raíz", el cual será el nodo padre, por defecto, de cualquier otro nodo que se vaya a añadir al árbol, y el primer nivel del árbol del espacio de búsqueda.

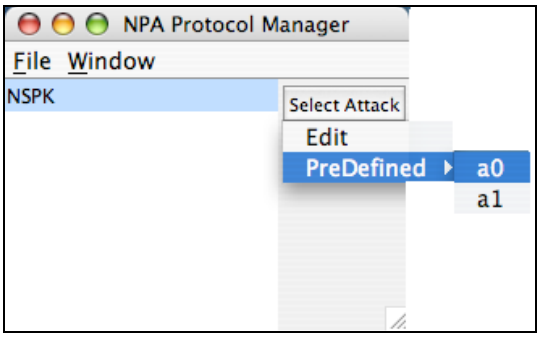


Figura 5: Selección de un estado de ataque del protocolo NSPK.

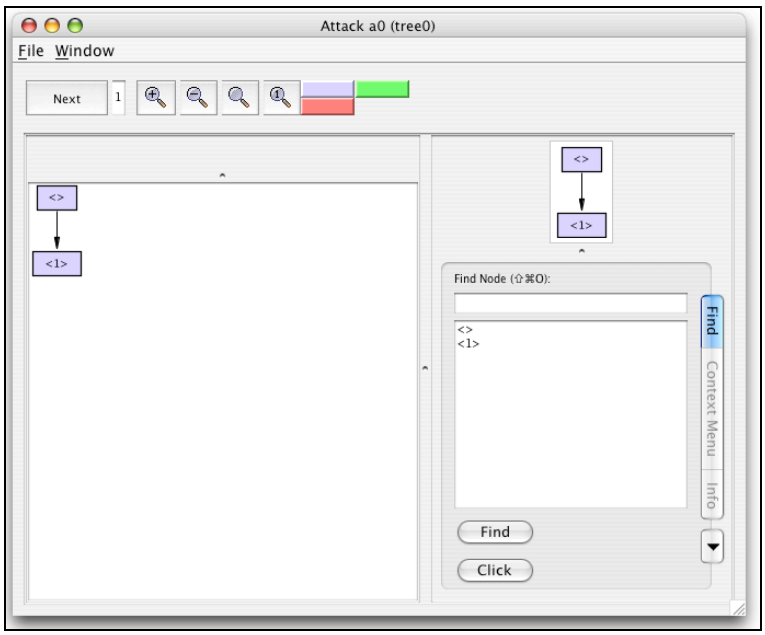


Figura 6: Árbol de búsqueda inicializado para el protocolo NSPK.

Para proseguir con el análisis, se pueden añadir uno o más niveles al árbol de búsqueda enviando una petición a Maude-NPA, a través de la IGU, para que los genere. En la ventana que muestra el árbol de búsqueda existe un botón llamado "Next" (ver Figura 7), el cual genera el número de niveles indicado por el usuario del espacio generado por la búsqueda hacia atrás indicado por el usuario. Cada nodo del árbol representa un estado obtenido por la búsqueda hacia atrás llevada a cabo por Maude-NPA. Los nodos pueden tener diferentes colores de fondo: lavanda si son nodos corrientes o verde si son estados iniciales, es decir, nodos solución. Si un nodo no tiene ningún sucesor en el árbol, es decir, Maude-NPA no ha encontrado en el espacio de búsqueda hacia atrás ningún estado que lo preceda, su color de fondo será rojo. La Figura 7 muestra el árbol del espacio de búsqueda para el protocolo NSPK cuando se ha encontrado un nodo solución.

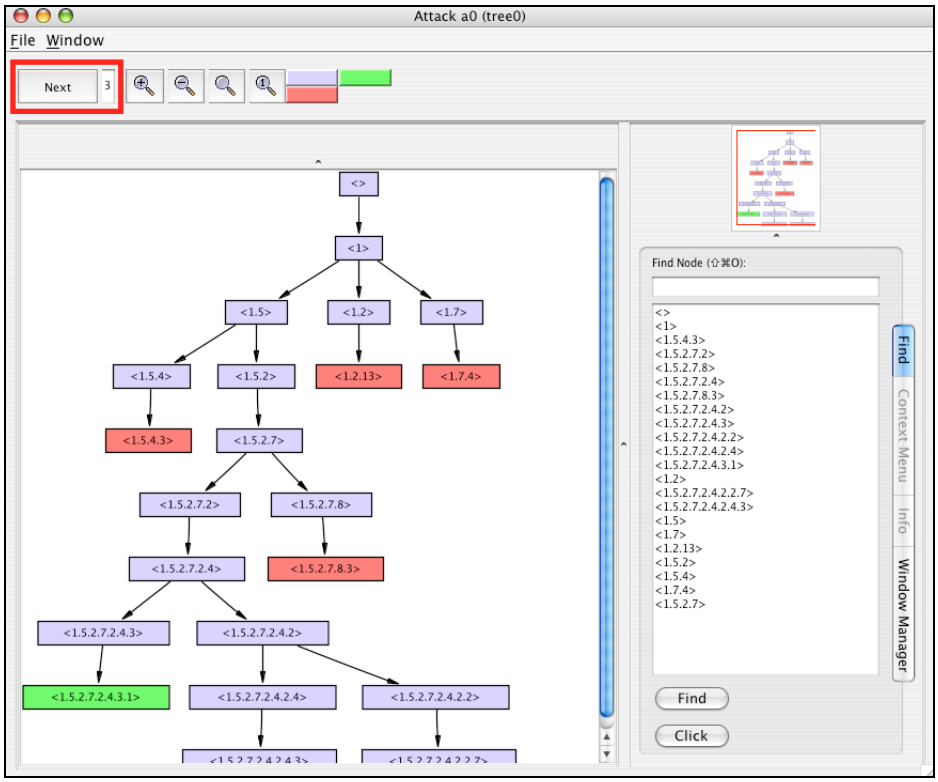


Figura 7: Árbol del espacio de búsqueda para el protocolo NSPK.

Maude-NPA genera para cada nodo la siguiente información (ver Sección 2):

1. Los strands actuales del estado,
2. el conocimiento del intruso,
3. la secuencia de mensajes, e
4. información adicional.

Cada nodo del árbol del espacio de búsqueda tiene un menú contextual asociado que permite al usuario consultar la información textual del estado generada por Maude-NPA, así como visualizar gráficamente esa información de estado, mediante una representación pictórica de los strands y del conocimiento del intruso. Para la visualización gráfica de los strands se ha seguido la representación gráfica original de los strands [13], introduciendo las modificaciones necesarias para representar la noción de tiempo de Maude-NPA. La Figura 8 muestra la representación gráfica de un estado de Maude-NPA, mediante la visualización de sus strands y del conocimiento del intruso.

Un strand se dibuja como un secuencia vertical de puntos unidos mediante una doble línea vertical. Cada punto, llamado también nodo, corresponde a un mensaje recibido o enviado por el strand. Tal y como se explica en la Sección 2, los strands permiten representar tanto el comportamiento de los participantes legítimos como el de los intrusos y, por consiguiente, también se muestran en la representación gráfica. Para diferenciar el strand de un participante legítimo del strand de un intruso, se utilizan diferentes colores: gris y negro para los strands de los participantes legítimos y verde claro y oscuro

para los strands del intruso. Se usan dos colores para cada tipo de strand con el propósito de representar la noción de tiempo: los colores claros para el pasado y el presente, y los colores oscuros para el futuro. La barra vertical usada en Maude-NPA para denotar la posición del tiempo se representa en esta IGU con una línea oblicua que cruza sobre la doble línea vertical que une los nodos del strand.

El conocimiento del intruso (t inI y t !inI) se integra en la representación gráfica mediante la utilización de diferentes colores para el mensaje pegado a cada nodo. Se usa el color rojo si el mensaje es conocido por el intruso (t inI) y el color negro si no lo es (t !inI). Para aquellos mensajes que no pertenecen a un strand en concreto, se utiliza únicamente un punto en lugar de una secuencia vertical de puntos.

Como conclusión de esta sección, se muestra cómo se representa textual y gráficamente el estado inicial asociado al protocolo NSPK en las Figuras 9 y 10, respectivamente.

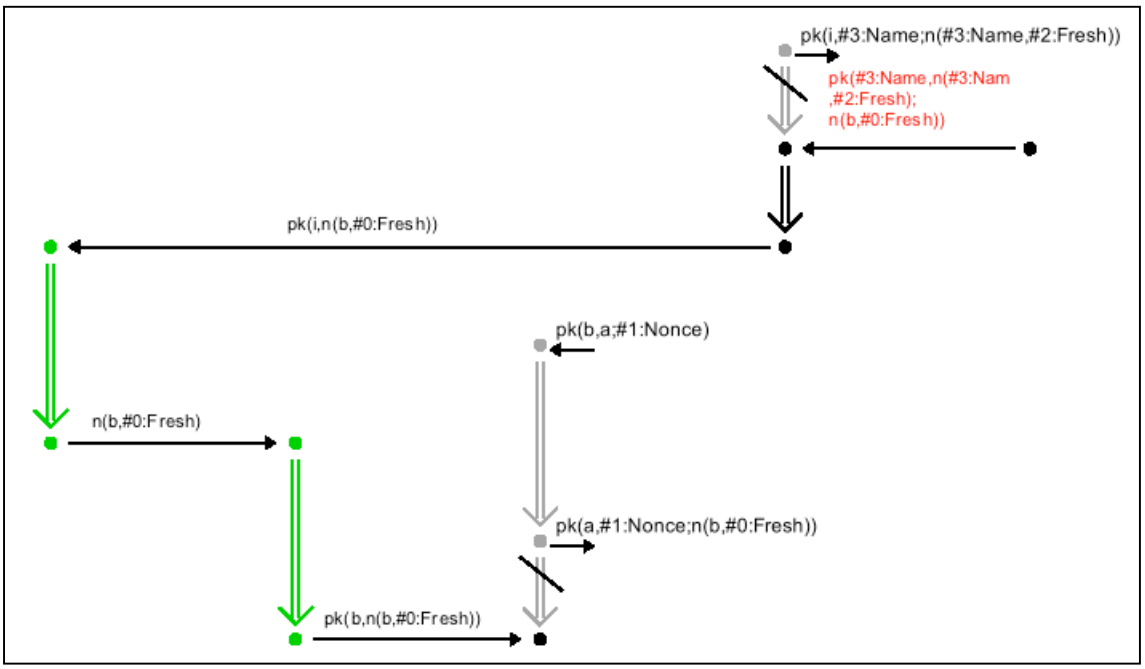


Figura 8: Representación gráfica de los strands de un estado.

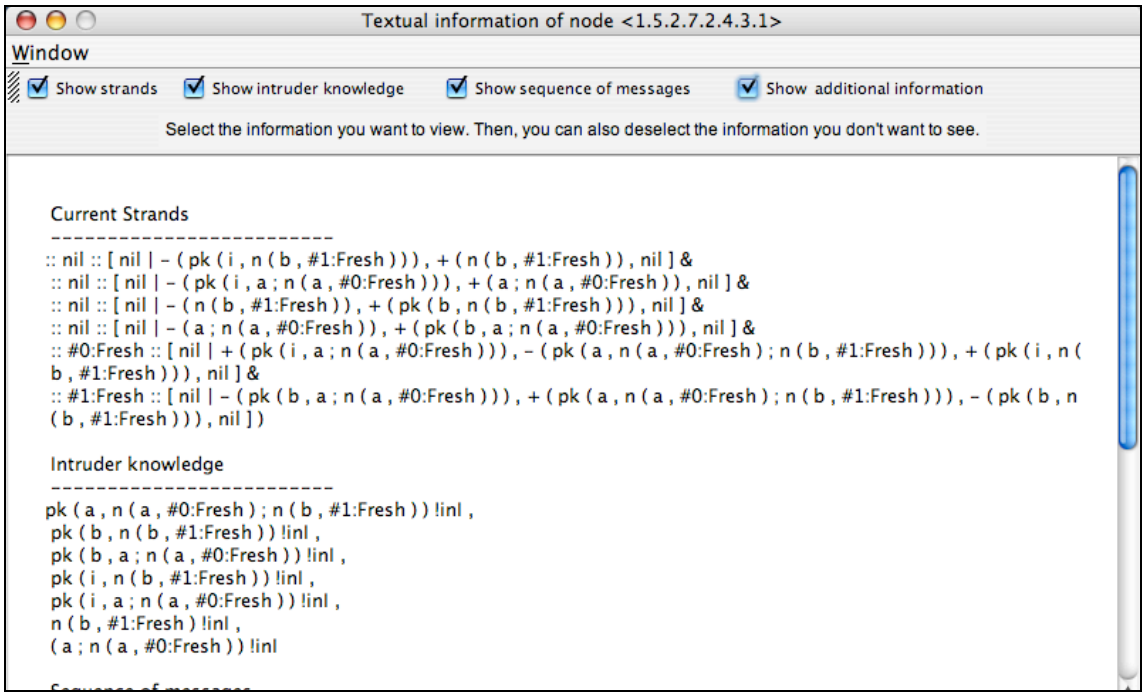


Figura 9: Información textual del estado inicial encontrado para el protocolo NSPK.

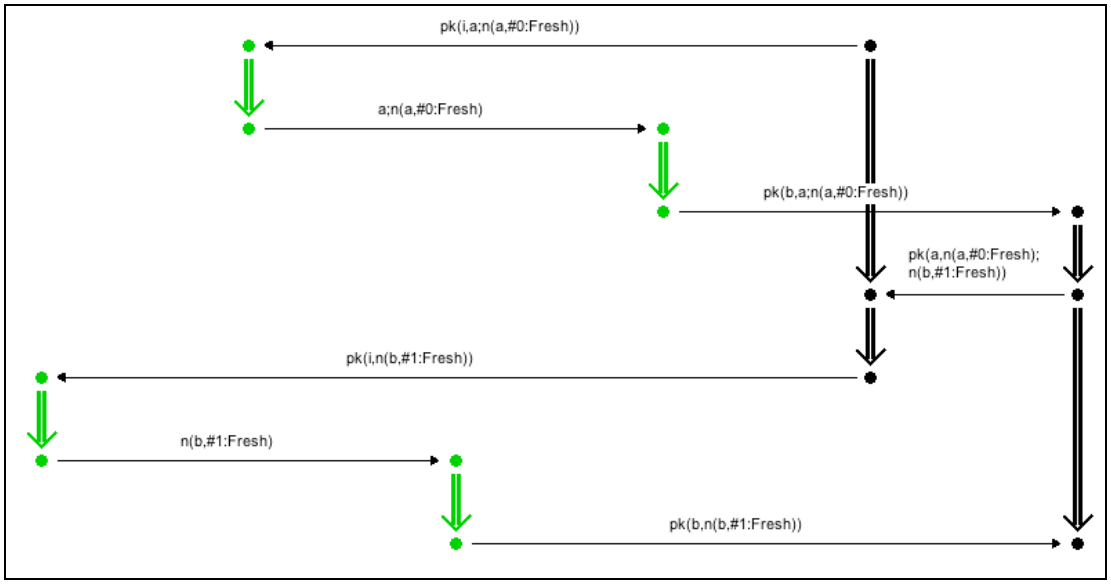


Figura 10: Visualización gráfica del estado inicial encontrado para el protocolo NSPK.

4. Interfaz.

4.1. Ventanas de la aplicación.

La IGU desarrollada está formada por cinco ventanas principales: (i) la ventana de inicio, (ii) la ventana para cargar el protocolo que se va a analizar, (iii) la ventana en la que se muestra el espacio de búsqueda generado por Maude-NPA, (iv) la ventana en la que se muestra textualmente la información de cada estado y (v) la ventana en la que se visualizan los strands de cada estado. Las cuatro primeras se explicarán con más detalle a continuación.

4.3. Ventana de inicio.

Esta es la primera ventana que se muestra cuando el usuario inicia la herramienta. Tal y como puede verse en la Figura 11, esta ventana consta de una barra de menú en la parte superior y de un panel de texto que ocupa el resto de la ventana. El menú "File" permitirá al usuario acceder a las ventanas para seleccionar el protocolo a analizar. En el panel, el usuario puede encontrar una breve descripción de la herramienta Maude-NPA, así como algunas indicaciones sobre cómo comenzar a utilizar la IGU.

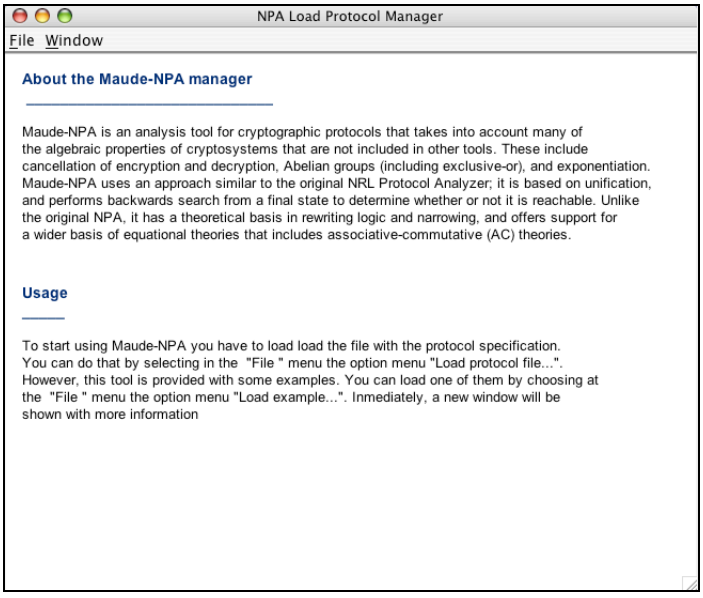


Figura 11: Ventana de inicio de la IGU

4.3. Ventana de carga de protocolo.

Mediante esta ventana (ver Figura 12), el usuario puede seleccionar y cargar en la herramienta la especificación del protocolo que desea analizar. Si el usuario ya ha analizado previamente esa misma especificación del protocolo, la herramienta habrá generado un archivo de texto que contiene la gramática asociada a ese protocolo. Esta gramática puede reutilizarse en los sucesivos análisis de ese protocolo, siempre y cuando su especificación no haya sido modificada.

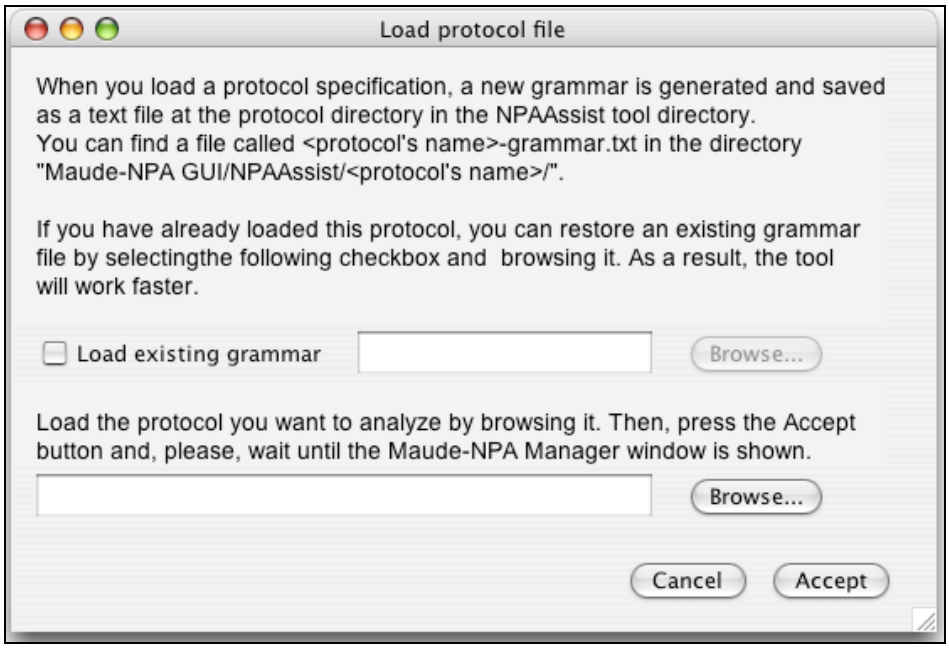


Figura 12: Ventana para cargar un protocolo.

Cuando no se carga ninguna gramática, Maude-NPA la generará automáticamente antes de iniciar el análisis. El único inconveniente es que la generación de la gramática requiere un cierto tiempo, por lo que se recomienda cargar una gramática existente, siempre que sea posible, para que el inicio del análisis sea más rápido. Adicionalmente se ofrecen algunos ejemplos de protocolos, para que el usuario pueda analizarlos y, de ese modo, familiarizarse con la herramienta. Esta funcionalidad se ofrece, también, a través de la ventana que se muestra en la Figura 13, que contiene una lista con los ejemplos de protocolo que se proporcionan.

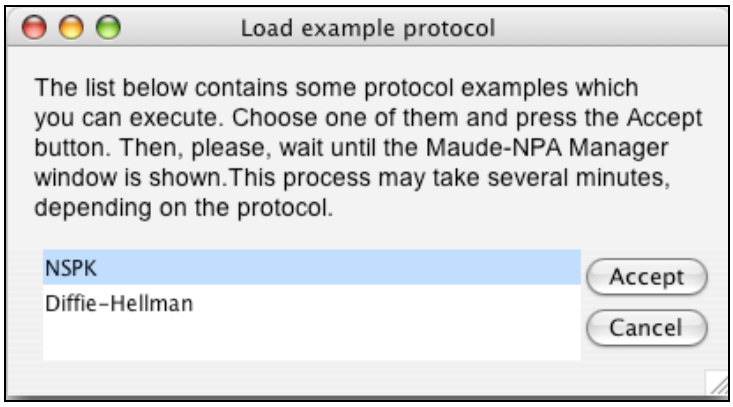


Figura 13: Ventana para cargar un protocolo de ejemplo.

4.4. Ventana para seleccionar un ataque del protocolo.

Esta ventana (ver Figura 14) permite, una vez se ha cargado el protocolo en la herramienta, seleccionar qué ataque del protocolo se desea analizar de los definidos en la especificación del protocolo (ver Sección 2.2 sobre cómo definir un ataque en el protocolo).

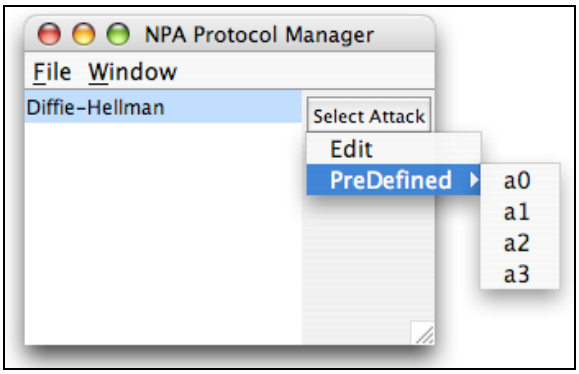


Figura 14: Ventana para seleccionar el ataque del protocolo a analizar.

4.5. Ventana para mostrar el espacio de búsqueda.

Esta es una de las principales ventanas de la IGU, puesto que es la encargada de representar gráficamente el espacio de búsqueda generado por Maude-NPA como un árbol cuyos nodos representan estados del espacio de búsqueda, tal y como se muestra en la Figura 15. Como se ha explicado anteriormente, cada uno de los nodos puede estar coloreado en color lavanda, verde o rojo. El color lavanda representa a estados corrientes, es decir, que no son estados iniciales. Por el contrario, el color verde indica que el estado al que representa es un estado inicial (ver Sección 2.2.2) y, por tanto, una solución al ataque del protocolo que se está analizando. El color rojo se

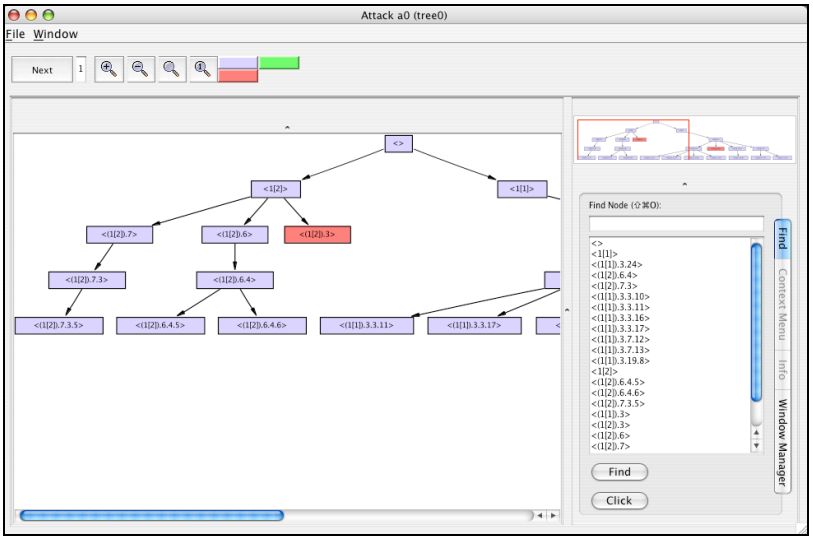


Figura 15: Ventana que representa gráficamente el espacio de búsqueda

utiliza para marcar los estados que Maude-NPA detecta como inalcanzables, es decir, aquellos que no tienen nodos sucesores (ver sección 2.2.3.) porque Maude-NPA no ha podido encontrar ningún estado que lo preceda en su búsqueda hacia atrás. Para cada uno de los nodos, puede obtenerse, en formato textual, la información asociada al estado que ese nodo representa, así como una representación gráfica.

Además, también permite al usuario ejecutar nuevos pasos de análisis y gestionar el conjunto de ventanas que el usuario ha abierto para explorar información más específica de cada uno de los estados del espacio de búsqueda.

4.6. Ventana para mostrar la información textual de un estado.

Como se mencionó en la Sección 2.2.1, Maude-NPA genera para cada estado cierta información: los strands correspondientes a ese estado, el conocimiento del intruso en ese momento de la ejecución del protocolo, la secuencia de mensajes intercambiados entre los participantes de la comunicación e información adicional.

La IGU permite mostrar toda esta información de forma textual para cada estado del espacio de búsqueda generado. La ventana que implementa esta funcionalidad puede verse en la Figura 16.

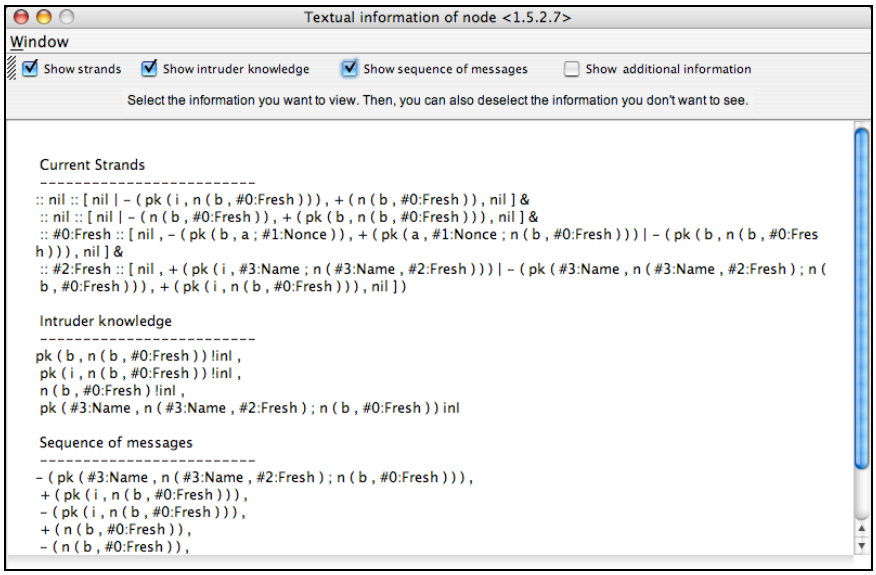


Figura 16: Ventana que muestra la información textual de un estado

En la parte superior de esta ventana, existen unos componentes gráficos que permiten al usuario elegir, concretamente, la información que desea ver. Dicha información se muestra en la parte central de la ventana.

5. Protocolos de clave simétrica con terceros participantes de confianza.

La principal característica de este tipo de protocolos es que nos encontramos con tres participantes:

1. Alice (el iniciador de la comunicación),
2. Bob, con quien quiere comunicarse Alice,
3. y un tercer participante "de confianza" que se encargará de las funciones de generación de una clave de confianza así como de su distribución, el Servidor.

5.1. Denning – Sacco Protocol.

Denning y Sacco sugieren arreglar el "freshness flaw" en el protocolo de Needham Shroeder con el uso de "Timestamps". La descripción informal del protocolo proporcionada en la sección 6.3.2 de [1], pág. 47, es la siguiente, donde E(K:M) significa que el mensaje M, encriptado con la clave K. Las distintas partes del mensaje M están separadas por comas:

(1) A → S : A,B
(2) S → A : E(Kas:B,Kab,T,E(Kbs:A,Kab,T))
(3) A → B : E(Kbs:A,Kab,T)

En el inicio del protocolo, Alice envía su nombre y el de Bob al Servidor. Éste le responde con el nombre de Bob, la clave que Alice y Bob deben de usar, el "timestamp" T y el mensaje para Bob, todo encriptado con la clave compartida entre Alice y el Servidor. Finalmente Alice envía el mensaje recibido para Bob directamente a él.

T es un "timestamp". En este caso hemos definido el "timestamp" como una variable TS de tipo Nonce. No hay necesidad de los intercambios extras entre Alice y Bob del protocolo de Needham-Schroeder. También utilizaremos la variable M de tipo Msg (mensaje) para especificar "E(Kbs:A,Kab,T)" en los strands de Alice. Codificaremos la clave de sesión Sk como Kab.

Los subtipos y operadores del protocolo, especificados en Maude-NPA quedarían de la siguiente manera:

```
--- Sort Information
 subsort Name Nonce  Key < Msg .
 subsort Masterkey  Sessionkey < Key .
 subsort SName UName < Name .
 subsort Name < Public .

--- Nonce operator
 op n : Name Fresh -> Nonce [frozen] .
 op t : Name Fresh -> Nonce [frozen] . ---Nonce del server

 --- User names
 ops a b i : -> UName .

 --- Server name
 op s : ->  SName .

 --- MKey
 op mkey : Name Name -> Masterkey [frozen] .

 --- Seskey
 op seskey : Name Name Nonce -> Sessionkey [frozen] .

 ---encrypt
 op e : Key Msg -> Msg [frozen] .
 op d : Key Msg -> Msg  [frozen] .

--- Concatenation
 op _;_ : Msg Msg -> Msg [frozen gather (e E)] .
```

Las propiedades algebraicas serían las siguientes:

```
fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
 protecting PROTOCOL-EXAMPLE-SYMBOLS .

 ------------------------------------------------------------
 --- Overwrite this module with the algebraic properties
 --- of your protocol
 ------------------------------------------------------------

 eq d(K:Key, e(K:Key, Z:Msg)) = Z:Msg [metadata "variant"] .
 eq e(K:Key, d(K:Key, Z:Msg)) = Z:Msg [metadata "variant"] .


endfm
```

Con estos operadores, un mensaje "E(Kas:A,B)" se escribe como "e(mkey(A,s),A;B)"

La especificación del comportamiento de Alice es la que sigue:

```
 --- Alice's Strand
  = :: nil ::
   [ nil  |  +(A ; B),
             -(e(mkey(A,s), B ; SK ; TS ; M)),
             +(M) , nil ]&
```

La especificación del comportamiento de Bob es la que sigue:

```
--- Bob's Strand
:: nil ::
[ nil | -(e(mkey(B,s), A ; SK ; TS)) , nil ] &
```

En la declaración del Servidor, utilizaremos para la representación del "Timestamp" el nonce generado por el servidor t(s,r') . Especificamos el comportamiento del Servidor como sigue:

```
:: r,r' ::
--- Server's Strand
[ nil | -(A ; B),
      +(e(mkey(A,s), B ;
        seskey(A , B , n(s,r)) ;
        t(s,r') ;
        e(mkey(B,s) , A ; seskey(A , B , n(s,r)) ;
        t(s,r'))))), nil ]
```

En este protocolo vamos a considerar tres configuraciones distintas que corresponden a las siguientes propiedades:

1. Una ejecución normal del protocolo (attack-state(0)).
2. Una ejecución donde el intruso averigua la clave de Alice y Bob generada por el servidor (attack-state(1)).
3. Una ejecución donde Bob ha completado el protocolo creyendo que habla con Alice pero no es así (attack-state(2)).

5.1.1. Attack-state(0). Ejecución normal.

La especificación del **attack-state(0)** en Maude-NPA sería:

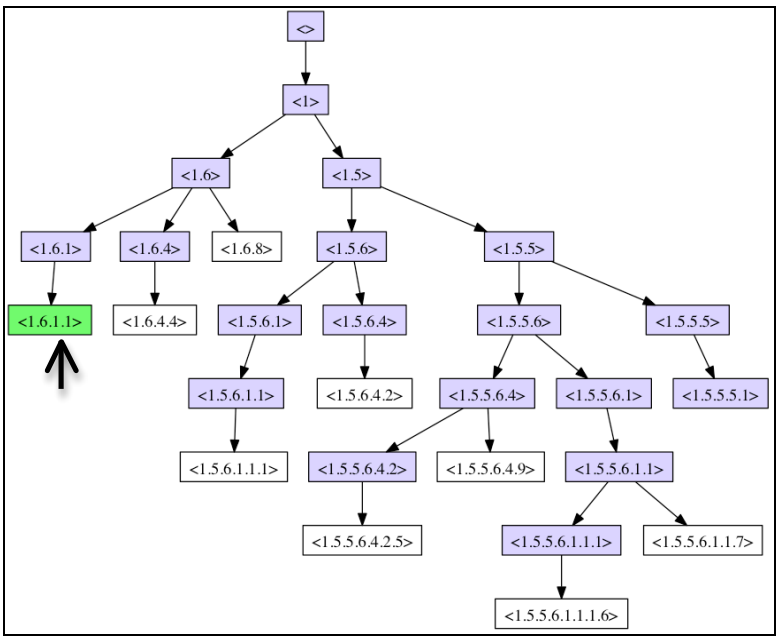
```
eq ATTACK-STATE(0) =
   :: nil ::
   [ nil , -(e(mkey(b,s), a ; SK ; TS)) | nil ]
   | | empty
   | | nil
   | | nil
   | | nil
[nonexec] .
```

Incluimos sólo el strand de Bob, sin requerir información sobre que conoce el intruso y la herramienta rellenará el resto.

Procederíamos a ejecutar el **attack-state(0)** mediante la herramienta gráfica de análisis.

Como podemos ver, la ejecución del attack-state(0) es una ejecución normal del protocolo.

Mostramos el árbol de búsqueda a continuación:



Donde podemos ver que existe una solución y por lo tanto una ejecución normal del protocolo.

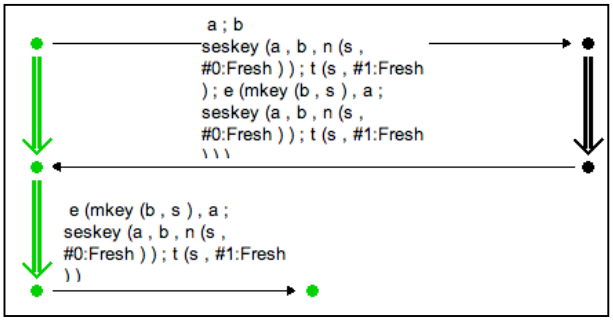
La secuencia de mensajes generados para la solución sería la siguiente:

```
generatedByIntruder ( a ; b ) ,
- ( a ; b ) ,
+ (e ( mkey ( a , s ) , b ; seskey ( a , b , n ( s , #0:Fresh ) ) ; t ( s , #1:Fresh );
   e ( mkey ( b , s ) , a ; seskey ( a , b , n ( s , #0:Fresh ) ) ;
    t ( s , #1:Fresh ) ) ) ) ,
+ ( a ; b ) ,
- ( e ( mkey ( a , s ) , b ; seskey ( a , b , n ( s , #0:Fresh ) ) ; t ( s , #1:Fresh );
   e ( mkey ( b , s ) , a ; seskey ( a , b , n ( s , #0:Fresh ) ) ;
    t ( s , #1:Fresh ) ) ) ) ,
 + ( e ( mkey ( b , s ) , a ; seskey ( a , b , n ( s , #0:Fresh ) ) ;
     t ( s , #1:Fresh ) ) ) ,
 - ( e ( mkey ( b , s ) , a ; seskey ( a , b , n ( s , #0:Fresh ) ) ;
    t ( s , #1:Fresh ) ) )
```

La representación en notación informal sería la siguiente

(1) A → S : A,B
(2) S → A : E(Kas:B,Kab,T,E(Kbs:A,Kab,T))
(3) A → B : E(Kbs:A,Kab,T)

La visualización gráfica de la solución obtenida sería la siguiente:



Donde podemos ver que se produce una ejecución normal del protocolo.

El strand de Alice generado por la herramienta gráfica sería el siguiente:

```
:: nil ::
---Alice's Strand
    [ nil | + ( a ; b ) ,
          - ( e ( mkey ( a , s ) , b ; seskey ( a , b , n ( s , #0:Fresh ) ) ;
              t ( s , #1:Fresh ) ;
            e ( mkey ( b , s ) , a ; seskey ( a , b , n ( s , #0:Fresh ) ) ;
            t ( s , #1:Fresh ) ) ) ) ,
          + ( e ( mkey ( b , s ) , a ; seskey ( a , b , n ( s , #0:Fresh ) ) ;
            t ( s , #1:Fresh ) ) ) , nil ] &
```

El strand de Bob generado por la herramienta gráfica sería el siguiente:

```
:: nil ::
---Bob's Strand
    [ nil | - ( e ( mkey ( b , s ) , a ; seskey ( a , b , n ( s , #0:Fresh ) ) ;
              t ( s , #1:Fresh ) ) ) , nil ] &
```

El strand del Servidor generado por la herramienta gráfica sería el siguiente:

```
:: #0:Fresh , #1:Fresh ::
---Server's Strand
    [ nil | - ( a ; b ) ,
          + ( e ( mkey ( a , s ) , b ; seskey ( a , b , n ( s , #0:Fresh ) ) ;
              t ( s , #1:Fresh ) ;
            e ( mkey ( b , s ) , a ; seskey ( a , b , n ( s , #0:Fresh ) ) ;
            t ( s , #1:Fresh ) ) ) ) , nil ]
```

5.1.2. Attack-state(1). Comprobación de secreto.

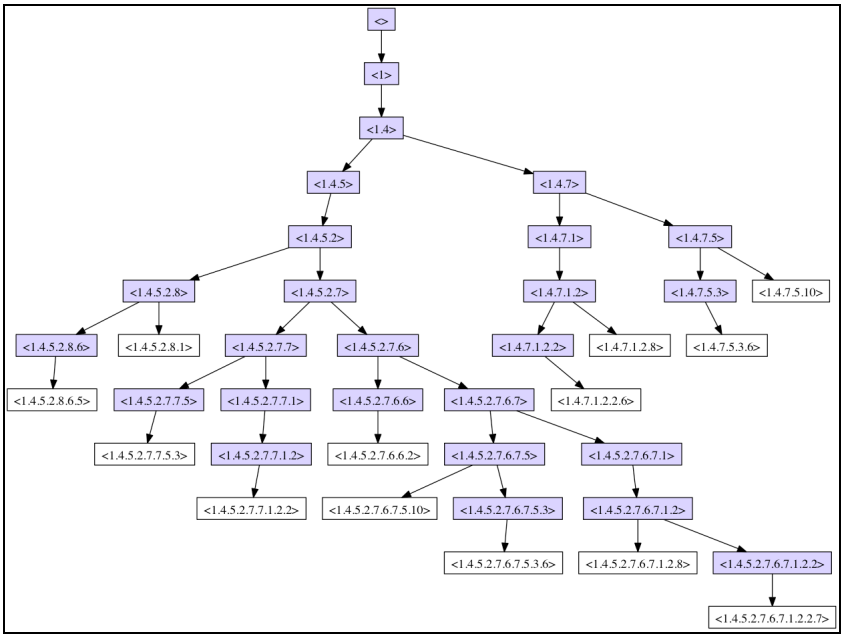
El attack-state(1) especificado en Maude-NPA sería el siguiente:

```
eq ATTACK-STATE(1) =
:: r ::
[ nil, -(a ; b),
     +(e(mkey(a,s), a ; SK ;  t(s,r') ; e(mkey(b,s) , a ; SK ; t(s,r'))))
       | nil ]
| | SK inI
| | nil
| | nil
| | nil
```

```
[nonexec] .
```

En este caso proporcionamos el strand del servidor e indicamos que el intruso podría conocer la clave de sesión entre Alice y Bob generada por el Servidor.

El árbol de búsqueda que se obtiene usando la herramienta gráfica sería el siguiente.



Donde podemos ver que no existe una solución. Por lo tanto es seguro ante un ataque de obtención de la clave de sesión.

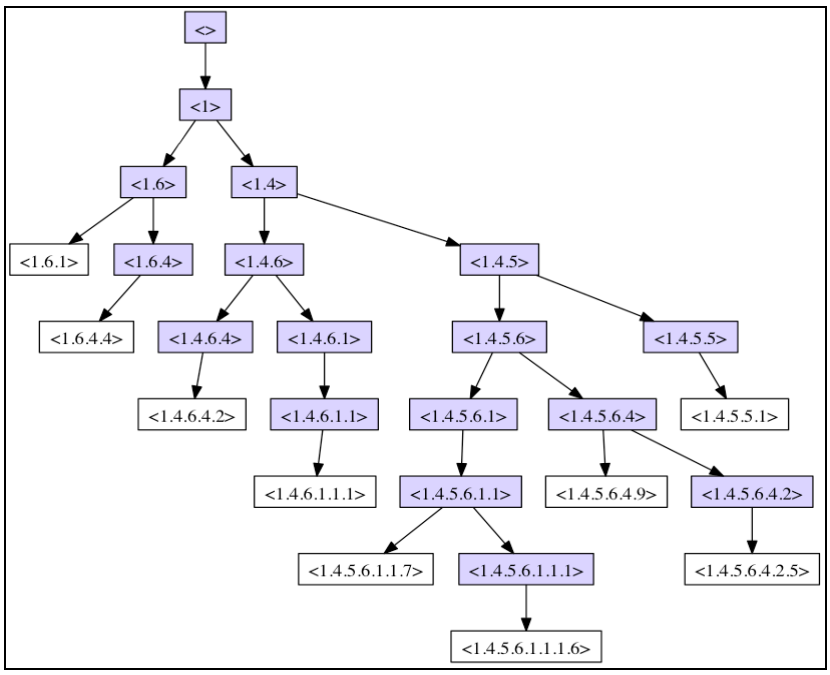
5.1.3. Attack-state(2). Comprobación de autenticación.

El attack-state(2) especificado en Maude-NPA sería el siguiente:

```
eq ATTACK-STATE(2) =
:: r ::
---Bob's Strand
   [ nil , -(e(mkey(b,s), a ; SK ; TS)) | nil ]
    | | empty
    | | nil
    | | nil
    | | never
   *** Pattern for authentication
   (:: R:FreshSet ::
   [ nil |  +(a ; b),
            -(e(mkey(a,s), b ; SK ; TS ; (e(mkey(B,s), A ; SK ; TS)))),
           +(M),  nil ]
    & S:StrandSet | | K:IntruderKnowledge)
 [nonexec] .
```

En este caso incluimos el strand de Bob y usaríamos el strand de Alice como patrón de autenticación.

El árbol de búsqueda que se obtiene al ejecutar la herramienta gráfica sería el siguiente:



Donde podemos ver que no existe una solución. Por lo tanto el protocolo es seguro ante un ataque de autenticación.

5.2. Otway-Rees Protocol.

El protocolo de **Otway-Rees** es un protocolo de autenticación de red diseñado para su uso en redes inseguras (por ejemplo, Internet).

Permite comunicarse entre dos participantes y probar su identidad el uno al otro, a la vez que prevenir el espionaje y los ataques de repetición ya que permite la detección de la modificación de la información transmitida.

La descripción informal del protocolo proporcionada en la Sección 6.3.3 de [1] , pág. 47, es la siguiente:

(1) A → B : M,A,B,E(Kas:Na,M,A,B)
(2) B → S : M,A,B,E(Kas:Na,M,A,B),E(Kbs:Nb,M,A,B)
(3) S → B : M,E(Kas:Na,Kab),E(Kbs:Nb,Kab)
(4) B → A : M,E(Kas:Na,Kab)

M es un nonce (un identificador de ejecución). En el mensaje 1 Alice envía a Bob el nonce M, los identificadores de Alice y Bob, y un mensaje encriptado leíble solo por el Servidor en la forma mostrada. Bob reenvía el mensaje al Servidor junto con una componente similar cifrada. El Servidor desencripta los componentes del mensaje y comprueba que el nonce M, y los identificadores de Alice y Bob son los mismos en ambos mensajes. Si es así, entonces genera una clave de sesión "Kab" y envía el mensaje 3 a Bob que reenvía parte del mensaje a Alice. Alice y Bob utilizarán la clave de sesión "Kab" solo si las componentes generadas por el Servidor contienen los nonces correctos "Na" y "Nb" respectivamente.

La especificación en Maude-NPA sería la siguiente:

```
--- Sort Information
 subsort Name Nonce  Key < Msg .
 subsort Masterkey  Sessionkey < Key .
 subsort SName UName < Name .
 subsort Name < Public .

--- Nonce operator
op n : Name Fresh -> Nonce [frozen] .
 --- User names
ops a b i : -> UName .

 --- Server name
 op s : ->  SName .

--- MKey
 op mkey : Name Name -> Masterkey [frozen comm] .

--- Seskey
 op seskey : Name Name Nonce -> Sessionkey [frozen] .

 ---encrypt
 op e  : Key Msg -> Msg [frozen] .
 op d : Key Msg -> Msg  [frozen] .

--- Concatenation
 op _;_ : Msg Msg -> Msg [frozen gather (e E)] .
```

Las propiedades algebraicas serían las siguientes:

```
fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
 protecting PROTOCOL-EXAMPLE-SYMBOLS .

 ------------------------------------------------------------
 --- Overwrite this module with the algebraic properties
 --- of your protocol
 ------------------------------------------------------------

 eq d(K:Key, e(K:Key, Z:Msg)) = Z:Msg [metadata "variant"] .
 eq e(K:Key, d(K:Key, Z:Msg)) = Z:Msg [metadata "variant"] .


endfm
```

En este protocolo existen tres roles: Alice, Bob y Servidor.

La especificación del comportamiento de Alice es la siguiente:

```
--- Alice's Strand.
  = :: r,rM ::
    [ nil | +(n(A,rM) ; A ; B ; e(mkey(A,s) , n(A,r) ; n(A,rM) ; A ; B)),
            -(n(A,rM) ; e(mkey(A,s) , n(A,r) ; SK)), nil ]
```

La especificación del comportamiento de Bob es la siguiente:

```
    :: r ::
  --- Bob's Strand.
    [ nil | -(NMA ; A ; B ; M1),
            +(NMA ; A ; B ; M1 ; e(mkey(B,s) , n(B,r) ; NMA ; A ; B)),
             -(NMA ; MA ; e(mkey(B,s) , n(B,r) ; SK)),
            +(NMA ; MA), nil ]
```

La especificación del comportamiento del Servidor es la siguiente:

```
    :: r ::
    --- Server's Strand
   [ nil |  -(NMA ; A ; B ;
              e(mkey(A,s), NA ; NMA ; A ; B) ;
              e(mkey(B,s) , NB ; NMA ; A ; B)),
           +(NMA ;
             e(mkey(A,s) , NA ; seskey(A , B , n(S,r))) ;
             e(mkey(B,s) , NB ; seskey(A , B , n(S,r)))) , nil]
```

En este protocolo vamos a considerar tres configuraciones distintas que corresponden a las siguientes propiedades:

1. Una ejecución normal del protocolo (attack-state(0)).
2. Una ejecución donde el intruso averigua la clave de Alice y Bob generada por el servidor (attack-state(1)).
3. Una ejecución donde Bob ha completado el protocolo creyendo que habla con Alice pero no es así (attack-state(2)).

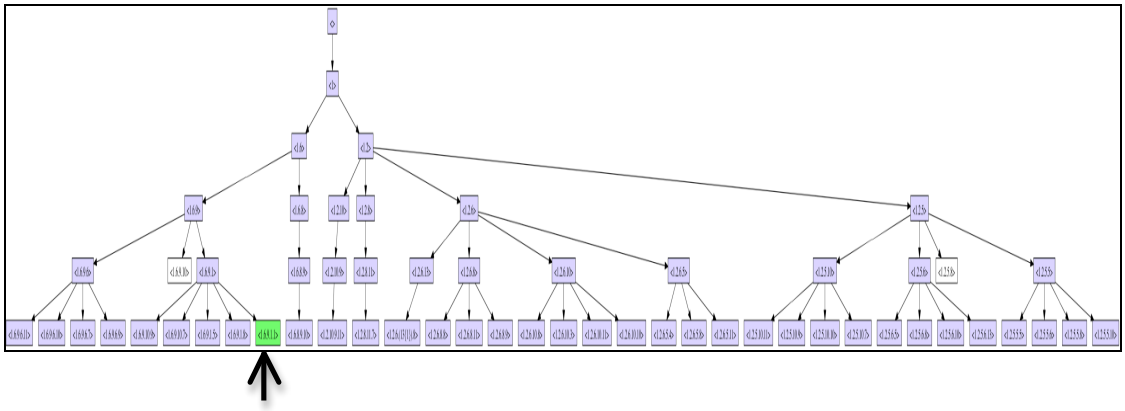
5.2.1. Attack-state(0). Ejecución normal.

La especificación del attack-state(0) en Maude-NPA sería la siguiente:

```
eq ATTACK-STATE(0) =
:: r,rM ::
--- Un estado con una ejecución normal del protocolo. añadimos el strand con el último -(M)
   [ nil ,  +(n(a,rM) ; a ; b ; e(mkey(a,s) , n(a,r) ; n(a,rM) ; a ; b)),
            -(n(a,rM) ; e(mkey(a,s) , n(a,r) ; SK)) | nil ]
    | | empty
    | | nil
    | | nil
    | | nil
[nonexec] .
```

Como podemos ver, la ejecución del attack-state(0) es una ejecución normal del protocolo. Se incluye el strand de Alice solamente.

Mostramos parte del árbol de búsqueda a continuación.



Donde podemos ver que existe una solución y por lo tanto una ejecución normal del protocolo.

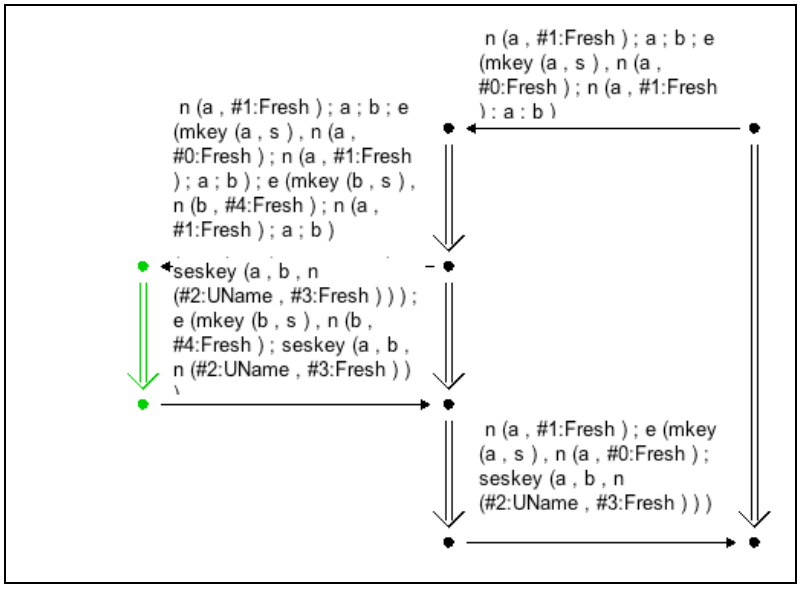
La secuencia de mensajes generados para la solución sería la siguiente:

```
+ ( n ( a , #1:Fresh ) ; a ; b ; e ( mkey ( a , s ) , n ( a , #0:Fresh ) ; n ( a , #1:Fresh ) ; a ; b ) ) ,
 - ( n ( a , #1:Fresh ) ; a ; b ; e ( mkey ( a , s ) , n ( a , #0:Fresh ) ; n ( a , #1:Fresh ) ; a ; b ) ) ,
+ ( n ( a , #1:Fresh ) ; a ; b ; e ( mkey ( a , s ) , n ( a , #0:Fresh ) ; n ( a , #1:Fresh ) ; a ; b ) ;
    e ( mkey ( b , s ) , n ( b , #4:Fresh ) ; n ( a , #1:Fresh ) ; a ; b ) ) ,
- ( n ( a , #1:Fresh ) ; a ; b ; e ( mkey ( a , s ) , n ( a , #0:Fresh ) ; n ( a , #1:Fresh ) ; a ; b ) ;
    e ( mkey ( b , s ) , n ( b , #4:Fresh ) ; n ( a , #1:Fresh ) ; a ; b ) ) ,
+ ( n ( a , #1:Fresh ) ; e ( mkey ( a , s ) , n ( a , #0:Fresh ) ;
    seskey ( a , b , n ( #2:UName , #3:Fresh ) ) ) ; e ( mkey ( b , s ) , n ( b , #4:Fresh ) ;
    seskey ( a , b , n ( #2:UName , #3:Fresh ) ) ) ) ,
 - ( n ( a , #1:Fresh ) ; e ( mkey ( a , s ) , n ( a , #0:Fresh ) ;
     seskey ( a , b , n ( #2:UName , #3:Fresh ) ) ) ; e ( mkey ( b , s ) , n ( b , #4:Fresh ) ;
     seskey ( a , b , n ( #2:UName , #3:Fresh ) ) ) ) ,
 + ( n ( a , #1:Fresh ) ; e ( mkey ( a , s ) , n ( a , #0:Fresh ) ;
     seskey ( a , b , n ( #2:UName , #3:Fresh ) ) ) ) ,
  - ( n ( a , #1:Fresh ) ; e ( mkey ( a , s ) , n ( a , #0:Fresh ) ;
      seskey ( a , b , n ( #2:UName , #3:Fresh ) ) ) )
```

Que se expresaría en notación informal de la siguiente manera:

(1) A → B : Na',A,B,E(Kas:Na,Na',A,B)
(2) B → Z(S) : Na',A,B,E(Kbs,Na,Na',A,B),E(Kbs,Nb,Na',A,B)
(3) Z(S) → B : Na',E(Kas,Na,Kab),E(Kbs,Nb,Kab)
(4) B → A : Na',E(Kas,Na,Kab)

La visualización gráfica de la solución obtenida sería la siguiente:



El strand de Alice generado por la herramienta gráfica sería el siguiente:

```
:: #0:Fresh , #1:Fresh ::
  [ nil | + ( n ( a , #1:Fresh ) ; a ; b ; e ( mkey ( a , s ) , n ( a , #0:Fresh ) ; n ( a , #1:Fresh ) ; a ; b ) ) ,
          - ( n ( a , #1:Fresh ) ; e ( mkey ( a , s ) , n ( a , #0:Fresh ) ;
              seskey ( a , b , n ( #2:UName , #3:Fresh ) ) ) ) , nil ]
```

El strand de Bob generado por la herramienta gráfica sería el siguiente:

```
:: #4:Fresh ::
[ nil | - ( n ( a , #1:Fresh ) ; a ; b ; e ( mkey ( a , s ) , n ( a , #0:Fresh ) ; n ( a , #1:Fresh ) ; a ; b ) ) ,
      + ( n ( a , #1:Fresh ) ; a ; b ; e ( mkey ( a , s ) , n ( a , #0:Fresh ) ; n ( a , #1:Fresh ) ; a ; b ) ;
        e ( mkey ( b , s ) , n ( b , #4:Fresh ) ; n ( a , #1:Fresh ) ; a ; b ) ) ,
       - ( n ( a , #1:Fresh ) ;
        e ( mkey ( a , s ) , n ( a , #0:Fresh ) ; seskey ( a , b , n ( #2:UName , #3:Fresh ) ) ) ;
        e ( mkey ( b , s ) , n ( b , #4:Fresh ) ; seskey ( a , b , n ( #2:UName , #3:Fresh ) ) ) ) ,
      + ( n ( a , #1:Fresh ) ; e ( mkey ( a , s ) , n ( a , #0:Fresh ) ;
        seskey ( a , b , n ( #2:UName , #3:Fresh ) ) ) ) , nil ] &
```

El strand del Servidor generado por la herramienta gráfica sería el siguiente:

```
:: #3:Fresh ::
  [ nil | - ( n ( a , #1:Fresh ) ; a ; b ; e ( mkey ( a , s ) , n ( a , #0:Fresh ) ; n ( a , #1:Fresh ) ; a ; b ) ;
            e ( mkey ( b , s ) , n ( b , #4:Fresh ) ; n ( a , #1:Fresh ) ; a ; b ) ) ,
          + ( n ( a , #1:Fresh ) ;
            e ( mkey ( a , s ) , n ( a , #0:Fresh ) ; seskey ( a , b , n ( #2:UName , #3:Fresh ) ) ) ;
            e ( mkey ( b , s ) , n ( b , #4:Fresh ) ; seskey ( a , b , n ( #2:UName , #3:Fresh ) ) ) ) , nil ] &
```

5.2.2. Attack-state(1). Comprobación de secreto.

Procederíamos a ejecutar el attack-state(1).

```
eq ATTACK-STATE(1) =
:: r,rM ::
--- Un estado con un secreto por descubrir
   [ nil ,  +(n(a,rM) ; a ; b ; e(mkey(a,s) , n(a,r) ; n(a,rM) ; a ; b)),
            -(n(a,rM) ; e(mkey(a,s) , n(a,r) ; SK)) | nil ]
    | | SK inI
    | | nil
    | | nil
    | | nil
[nonexec] .
```

En este caso indicamos que el intruso podría conocer la clave de sesión entre Alice y Bob generada por el Servidor.

En la ejecución de este ataque no se ha obtenido ninguna solución ni se ha podido finalizar la ejecución ya que el numero de estado ha sido excesivo y la herramienta no ha podido manejar el protocolo.

5.2.3. Attack-state(2). Comprobación de autenticación.

Procederíamos a ejecutar el attack-state(2) mediante la herramienta gráfica de análisis.

Definiríamos los siguientes mensajes:

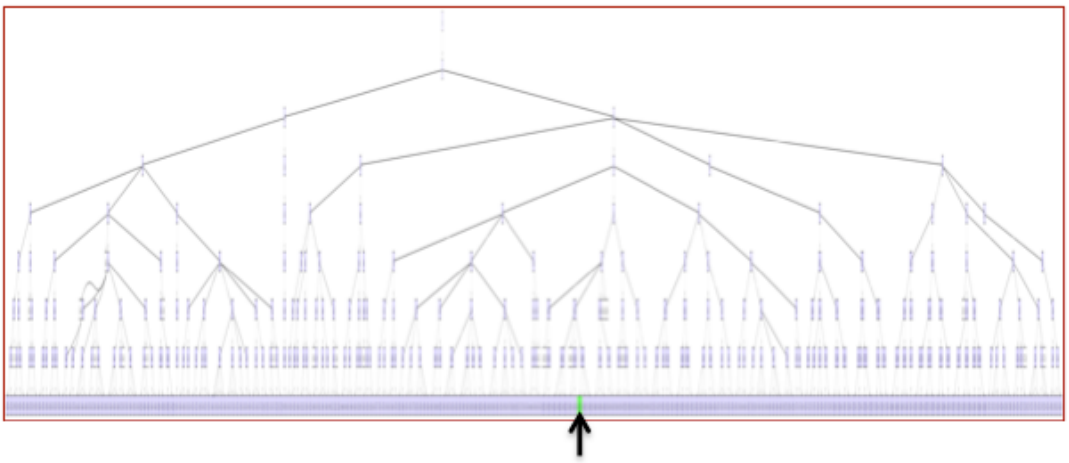
- MB1 como "e(mkey(B,s) , n(B,r) ; NMA ; A ; B)".
- MB2 como "e(mkey(B,s) , n(B,r) ; SK)"

```
eq ATTACK-STATE(2) =
--- Un estado con una ejecución normal pero con patron de autenticación.
:: r,rM ::
   [ nil ,  +(n(a,rM) ; a ; b ; e(mkey(a,s) , n(a,r) ; n(a,rM) ; a ; b)),
           -(n(a,rM) ; e(mkey(a,s) , n(a,r) ; SK)) | nil ]
| | empty
| | nil
| | nil
| | never
*** Pattern for authentication
   (:: r' ::
   [ nil | -(n(a,rM) ; a ; b ; e(mkey(a,s) , n(a,r) ; n(a,rM) ; a ; b)),
           +(n(a,rM) ; a ; b ; e(mkey(a,s) , n(A,r) ; n(A,rM) ; a ; b) ; MB1),
            -(n(a,rM) ; e(mkey(a,s) , n(a,r) ; SK) ; MB2),
           +(n(a,rM) ; e(mkey(a,s) , n(a,r) ; SK)), nil ]
    & S:StrandSet | | K:IntruderKnowledge)

[nonexec] .
```

En este caso usaríamos el strand de Alice como patrón de autenticación.

Mostramos parte del árbol de búsqueda a continuación.



Donde podemos ver que existe una solución. Esto significa que el protocolo no es seguro ante un ataque de autenticación.

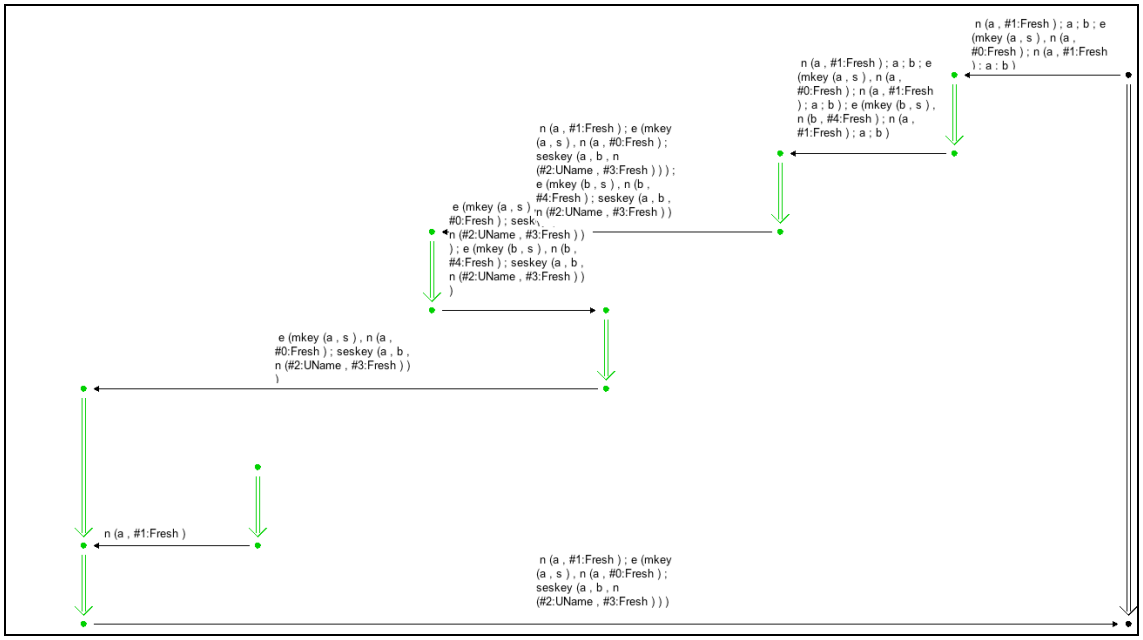
La secuencia de mensajes generados para la solución sería la siguiente:

```
+ ( n ( a , #1:Fresh ) ; a ; b ;
    e ( mkey ( a , s ) , n ( a , #0:Fresh ) ;  n ( a , #1:Fresh ) ; a ; b ) ) ,
 - ( n ( a , #1:Fresh ) ; a ; b ;
    e ( mkey ( a , s ) , n ( a , #0:Fresh ) ; n ( a , #1:Fresh ) ; a ; b ) ) ,
 + ( n ( a , #1:Fresh ) ; a ; b ;
     e ( mkey ( a , s ) , n ( a , #0:Fresh ) ; n ( a , #1:Fresh ) ; a ; b ) ;
     e ( mkey ( b , s ) , n ( b , #4:Fresh ) ; n ( a , #1:Fresh ) ; a ; b ) ) ,
  - ( n ( a , #1:Fresh ) ; a ; b ;
     e ( mkey ( a , s ) , n ( a , #0:Fresh ) ; n ( a , #1:Fresh ) ; a ; b ) ;
     e ( mkey ( b , s ) , n ( b , #4:Fresh ) ; n ( a , #1:Fresh ) ; a ; b ) ) ,
 + ( n ( a , #1:Fresh ) ;
     e ( mkey ( a , s ) , n ( a , #0:Fresh ) ;
     seskey ( a , b , n ( #2:UName , #3:Fresh ) ) ) ;
     e ( mkey ( b , s ) , n ( b , #4:Fresh ) ;
     seskey ( a , b , n ( #2:UName , #3:Fresh ) ) ) ) ,
 - ( n ( a , #1:Fresh ) ;
    e ( mkey ( a , s ) , n ( a , #0:Fresh ) ;
    seskey ( a , b , n ( #2:UName , #3:Fresh ) ) ) ;
    e ( mkey ( b , s ) , n ( b , #4:Fresh ) ;
    seskey ( a , b , n ( #2:UName , #3:Fresh ) ) ) ) ,
 + ( e ( mkey ( a , s ) , n ( a , #0:Fresh ) ;
     seskey ( a , b , n ( #2:UName , #3:Fresh ) ) ) ;
     e ( mkey ( b , s ) , n ( b , #4:Fresh ) ;
     seskey ( a , b , n ( #2:UName , #3:Fresh ) ) ) ) ,
 - ( e ( mkey ( a , s ) , n ( a , #0:Fresh ) ;
     seskey ( a , b , n ( #2:UName , #3:Fresh ) ) ) ;
     e ( mkey ( b , s ) , n ( b , #4:Fresh ) ;
     seskey ( a , b , n ( #2:UName , #3:Fresh ) ) ) ) ,
 + ( e ( mkey ( a , s ) , n ( a , #0:Fresh ) ;
     seskey ( a , b , n ( #2:UName , #3:Fresh ) ) ) ) ,
 - ( n ( a , #1:Fresh ) ; a ; b ;
    e ( mkey ( a , s ) , n ( a , #0:Fresh ) ; n ( a , #1:Fresh ) ; a ; b ) ) ,
 + ( n ( a , #1:Fresh ) ) ,
  - ( e ( mkey ( a , s ) , n ( a , #0:Fresh ) ;
    seskey ( a , b , n ( #2:UName , #3:Fresh ) ) ) ) ,
  - ( n ( a , #1:Fresh ) ) ,
 + ( n ( a , #1:Fresh ) ;
     e ( mkey ( a , s ) , n ( a , #0:Fresh ) ;
     seskey ( a , b , n ( #2:UName , #3:Fresh ) ) ) ) ,
 - ( n ( a , #1:Fresh ) ;
    e ( mkey ( a , s ) , n ( a , #0:Fresh ) ;
    seskey ( a , b , n ( #2:UName , #3:Fresh ) ) ) )
```

Que se expresaría en notación informal de la siguiente manera:

(1)	A → Z(B) : Na',A,B,E(Kas:Na,Na',A,B)
(2)	Z(B) → Z(S) : Na',A,B,E(Kbs,Na,Na',A,B), E(Kbs,Nb,Na',A,B)
(3)	Z(S) → Z(B) : Na',E(Kas,Na,Kab),E(Kbs,Nb,Kab)
(4)	Z(B) → Z(S) : E(Kas,Na,Kab),E(Kbs,Nb,Kab)
(5)	Z(S) → Z(B) : E(Kas,Na,Kab),
(6)	Z(S) → Z(B) : Na'
(7)	Z(B) → A : Na',E(Kas,Na,Kab)

La visualización gráfica de la solución obtenida sería la siguiente:



5.3. Amended Needham Schroeder Protocol.

Este protocolo es el resultado que proponen Needham y Schroeder como una solución para el Protocolo original, la cual evita el ataque que compromete a la clave. La descripción informal del protocolo proporcionada en la sección 6.3.4 de [1] , pág. 48, es la siguiente:

(1)	A → B : A
(2)	B → A : E(Kbs:A,Nb0)
(3)	A → S : A,B,Na,E(Kbs:A,Nb0)
(4)	S → A : E(Kas:Na,B,Kab,E(Kbs:Kab,Nb0,A))
(5)	A → B : E(Kbs:Kab,Nb0,A)
(6)	B → A : E(Kab:Nb)
(7)	A → B : E(Kab:Nb-1)

Al comienzo, en el mensaje 1, Alice le envía a Bob una solicitud para comunicarse, enviándole su identidad. Bob le responde, en el mensaje 2, enviándole un Nonce (Nb0) encriptado bajo su clave con el Servidor (Kbs). En el mensaje 3, Alice envía un mensaje al servidor identificándose a sí misma ante Bob, diciéndole al servidor que ella quiere comunicarse con Bob. Seguidamente, el servidor, en el mensaje 4, genera la clave de sesión (Kab) y le devuelve a Alice una copia encriptada bajo la clave Kbs de Alice que transmite a Bob y también una copia a Alice. Ya que Alice puede solicitar claves de varias personas, el nonce asegura a Alice que el mensaje es reciente y que el Servidor está respondiendo a ese mensaje en particular y la inclusión del nombre del Bob le dice a Alice que es con ella con quien se quiere compartir la clave. Después, Alice envía la clave, en el mensaje 5, para que Bob pueda descifrar, con la clave que comparte con el servidor, el mensaje y así autenticar los datos. Posteriormente, en el mensaje 6, Bob envía a Alice un nonce encriptado bajo la clave de sesión generada por el servidor para demostrar que él tiene la clave. Por último, en el mensaje 7, Alice lleva a cabo una operación en el Nonce (Nb-1), se re-codifica y envía de vuelta la verificación de que ella todavía está viva y que ella tiene la clave de sesión generada por el Servidor.

Los subtipos y operadores del protocolo, especificados en Maude-NPA quedarían de la siguiente manera:

```
--- Sort Information
 subsort Name Nonce  Key < Msg .
 subsort Masterkey  Sessionkey < Key .
 subsort SName UName < Name .
 subsort Name < Public .

--- Nonce operator
 op n : Name Fresh -> Nonce [frozen] .

 --- User names
 ops a b i : -> UName .

 --- Server name
 op s : ->  SName .


 --- MKey
 op mkey : Name Name -> Masterkey [frozen] .

 --- Seskey
 op seskey : Name Name Nonce -> Sessionkey [frozen] .

 ---encrypt
 op e  : Key Msg -> Msg [frozen] .
 op d : Key Msg -> Msg  [frozen] .

 --- decrements
 op dec : Msg -> Msg [frozen] .


 --- Concatenation
 op _;_ : Msg Msg -> Msg [frozen gather (e E)] .
```

Las propiedades algebraicas serían las siguientes:

```
fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
  protecting PROTOCOL-EXAMPLE-SYMBOLS .

  ------------------------------------------------------------
  --- Overwrite this module with the algebraic properties
  --- of your protocol
  ------------------------------------------------------------

 eq d(K:Key, e(K:Key, Z:Msg)) = Z:Msg [metadata "variant"] .
 eq e(K:Key, d(K:Key, Z:Msg)) = Z:Msg [metadata "variant"] .


endfm
```

En este protocolo existen tres roles: Alice, Bob y Servidor.

La especificación del comportamiento de Alice es la siguiente:

```
   = :: r ::
--- Alice's Strand
    [ nil | +(A),
            -(M),
            +(A ; B ; n(A,r) ; M ) ,
            -(e (mkey(A, s), n(A,r) ; B ; seskey(A,B,NS)) ; N) ,
            + (N) ,
            -(e(SK , NB)) ,
            + (e(SK, dec(NB))) , nil ]
```

La especificación de la clave de sesión es "seskey(A,B,NS))" ya que Alice puede verla, a diferencia de Bob que no puede y por eso la especificaremos en su strand como "SK".

La especificación del comportamiento de Bob es la siguiente:

```
    :: r , r' ::
--- Bob's Strand
    [ nil | -(A),
            +(e(mkey(B,s), A ; n(B,r))),
            -(e(mkey(B,s), SK ; n(B,r) ;  A)),
            +(e(SK, n(B,r'))) ,
            -(e(SK, dec(n(B,r')))),  nil ]
```

La especificación del comportamiento del Servidor es la siguiente:

```
    :: r ::
--- Server's Strand
    [ nil | -( A ; B ; NA ; e(mkey(B,s), A ; NB )) ,
            +( e(mkey(A,s) , NA ; B ; seskey(A,B,n(S,r))) ;
              e(mkey(B,s), seskey(A,B, n(S,r)) ; NB0 ; A )) ,  nil ]
```

En este protocolo vamos a considerar dos configuraciones distintas que corresponden a las siguientes propiedades:

1. Una ejecución normal del protocolo (attack-state(0)).
2. Una ejecución donde el intruso averigua la clave de Alice y Bob generada por el servidor (attack-state(1)).

Al ejecutar la primera de las dos configuraciones observamos que el intruso engaña a los participantes haciéndoles creer que están comunicándose con el Servidor, cuando realmente lo están haciendo con él. Con lo cual, el protocolo no sería seguro antes una comprobación de autenticación, ya que se puede suplantar al Servidor. Debido a eso, no se ejecuta dicha configuración.

5.3.1. Attack-state(0). Ejecución normal.

Especificaríamos el attack-state(0) en Maude-NPA de la siguiente manera:

```
eq ATTACK-STATE(0) =
   :: r, r' ::
--- Bob's Strand
   [ nil ,-(a),
        +(e(mkey(b,s), a ; n(b,r))),
        -(e(mkey(b,s), SK ; n(b,r) ;  a)),
        +(e(SK, n(b,r'))) ,
        -(e(SK, dec(n(b,r')))) | nil ]
   | | empty
   | | nil
   | | nil
   | | nil
 [nonexec] .
```

Incluimos sólo el strand de Bob, sin requerir información sobre que conoce el intruso y la herramienta rellenará el resto.

Como podemos ver, la ejecución del attack-state(0) es una ejecución normal del protocolo.. Usaremos también el operador sucesor "**op** p : Msg -> Msg ."

Procederíamos a ejecutar el ° mediante la herramienta gráfica de análisis.

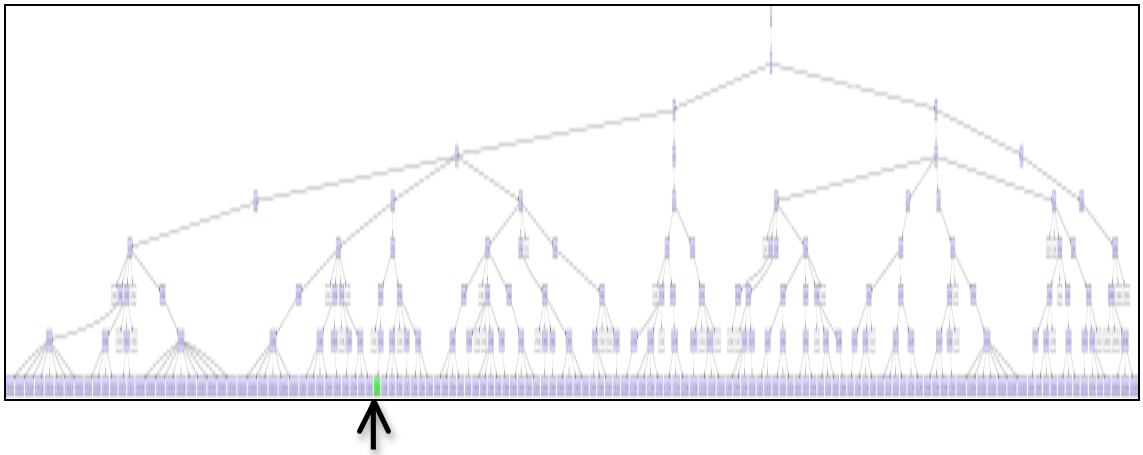
La secuencia de mensajes generados para la solución sería la siguiente:

```
 - ( a ) ,
+ ( e ( mkey ( b , s ) , a ; n ( b , #0:Fresh ) ) ) ,
+ ( a ) ,
 - ( e ( mkey ( b , s ) , a ; n ( b , #0:Fresh ) ) ) ,
+ ( a ; b ; n ( a , #4:Fresh ) ; e ( mkey ( b , s ) , a ; n ( b , #0:Fresh ) ) ) ,
 - ( a ; b ; n ( a , #4:Fresh ) ; e ( mkey ( b , s ) , a ; n ( b , #0:Fresh ) ) ) ,
+ ( e ( mkey ( a , s ) , n ( a , #4:Fresh ) ; b ;
    seskey ( a , b , n ( #2:Name , #3:Fresh ) ) ; e ( mkey ( b , s ) ,
    seskey ( a , b , n ( #2:Name , #3:Fresh ) ) ; n ( b , #0:Fresh ) ; a ) ) ) ,
- ( e ( mkey ( a , s ) , n ( a , #4:Fresh ) ; b ;
    seskey ( a , b , n ( #2:Name , #3:Fresh ) ) ; e ( mkey ( b , s ) ,
    seskey ( a , b , n ( #2:Name , #3:Fresh ) ) ; n ( b , #0:Fresh ) ; a ) ) ) ,
+ ( e ( mkey ( b , s ) , seskey ( a , b , n ( #2:Name , #3:Fresh ) ) ; n ( b , #0:Fresh ) ; a ) ) ,
 - ( e ( mkey ( b , s ) , seskey ( a , b , n ( #2:Name , #3:Fresh ) ) ; n ( b , #0:Fresh ) ; a ) ) ,
+ ( e ( seskey ( a , b , n ( #2:Name , #3:Fresh ) ) , n ( b , #1:Fresh ) ) ) ,
 - ( e ( seskey ( a , b , n ( #2:Name , #3:Fresh ) ) , n ( b , #1:Fresh ) ) ) ,
+ ( e ( seskey ( a , b , n ( #2:Name , #3:Fresh ) ) , dec ( n ( b , #1:Fresh ) ) ) ) ,
 - ( e ( seskey ( a , b , n ( #2:Name , #3:Fresh ) ) , dec ( n ( b , #1:Fresh ) ) ) )
```

La expresión de esta secuencia de mensajes en notación informal sería la siguiente:

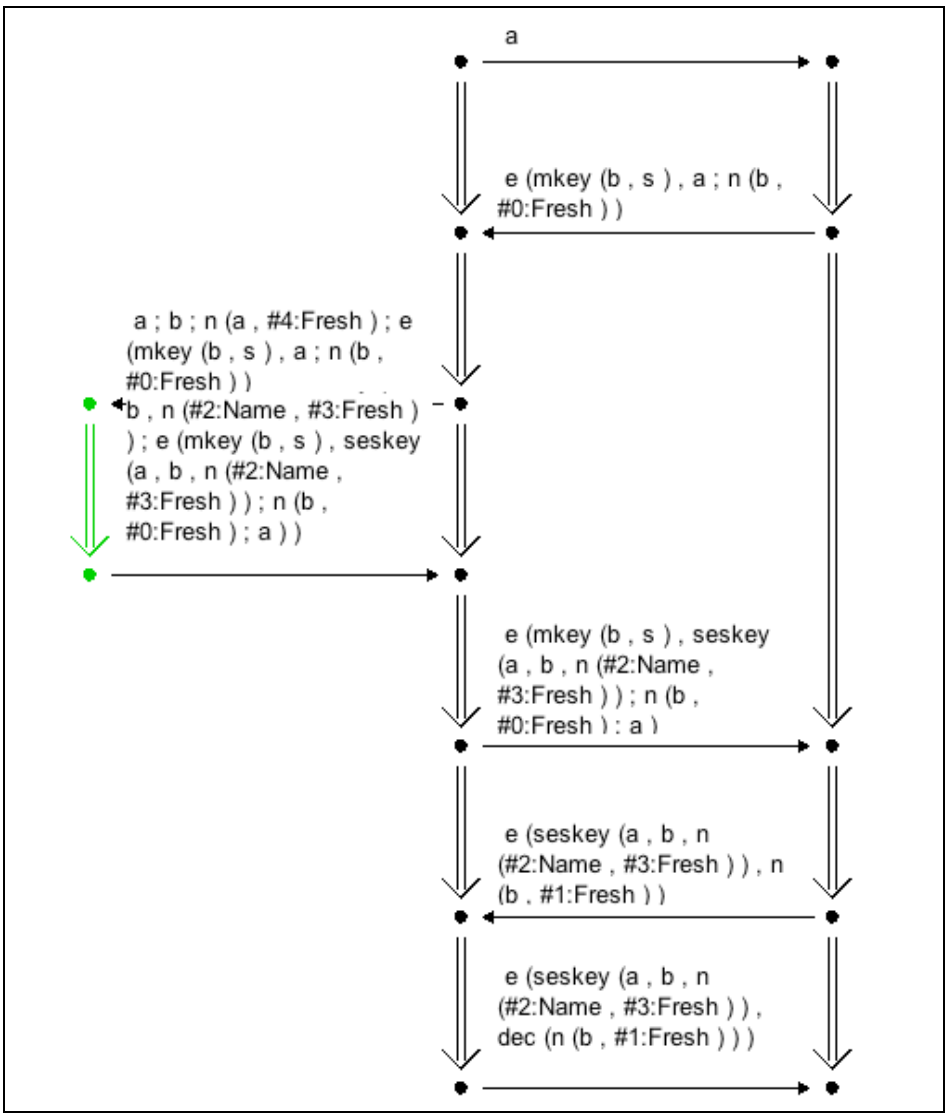
(1) A → B : A
(2) B → A : E(Kbs:A,Nb0)
(3) A → Z(S) : A,B,Na,E(Kbs:A,Nb0)
(4) Z(S) → A : E(Kas:Na,B,Kab,E(Kbs:Kab,Nb0,A))
(5) A → B : E(Kbs:Kab,Nb0,A)
(6) B → A : E(Kab:Nb)
(7) A → B : E(Kab:Nb-1)

Mostramos parte del árbol de búsqueda a continuación.



Donde podemos ver que existe una solución.

Y la visualización gráfica de la solución obtenida sería la siguiente:



Donde podemos ver que el protocolo no es seguro. Podemos observar que el protocolo no es seguro ante una comprobación de autenticación ya que los participantes piensan que están hablando con el Servidor cuando realmente lo están haciendo con el intruso.

El strand del Servidor generado por la herramienta gráfica sería el siguiente:

```
:: #3:Fresh ::
 [ nil | - ( a ; b ; n ( a , #4:Fresh ) ; e ( mkey ( b , s ) , a ; n ( b , #0:Fresh ) ) ) ,
         + ( e ( mkey ( a , s ) , n ( a , #4:Fresh ) ; b ;
             seskey ( a , b , n ( #2:Name , #3:Fresh ) ) ; e ( mkey ( b , s ) ,
             seskey ( a , b , n ( #2:Name , #3:Fresh ) ) ; n ( b , #0:Fresh ) ; a ) ) ) , nil ] &
```

El strand de Alice generado por la herramienta gráfica sería el siguente:

```
:: #4:Fresh ::
  [ nil | + ( a ) ,
          - ( e ( mkey ( b , s ) , a ; n ( b , #0:Fresh ) ) ) ,
          + ( a ; b ; n ( a , #4:Fresh ) ; e ( mkey ( b , s ) , a ; n ( b , #0:Fresh ) ) ) ,
          - ( e ( mkey ( a , s ) , n ( a , #4:Fresh ) ; b ; seskey ( a , b , n ( #2:Name , #3:Fresh ) ) ;
              e ( mkey ( b , s ) , seskey ( a , b , n ( #2:Name , #3:Fresh ) ) ; n ( b , #0:Fresh ) ; a ) ) ) ,
          + ( e ( mkey ( b , s ) , seskey ( a , b , n ( #2:Name , #3:Fresh ) ) ; n ( b , #0:Fresh ) ; a ) ) ,
          - ( e ( seskey ( a , b , n ( #2:Name , #3:Fresh ) ) , n ( b , #1:Fresh ) ) ) ,
          + ( e ( seskey ( a , b , n ( #2:Name , #3:Fresh ) ) , dec ( n ( b , #1:Fresh ) ) ) ) , nil ] &
```

El strand de Bob generado por la herramienta gráfica sería el siguente:

```
:: #0:Fresh , #1:Fresh ::
  [ nil | - ( a ) ,
          + ( e ( mkey ( b , s ) , a ; n ( b , #0:Fresh ) ) ) ,
          - ( e ( mkey ( b , s ) , seskey ( a , b , n ( #2:Name , #3:Fresh ) ) ; n ( b , #0:Fresh ) ; a ) ) ,
          + ( e ( seskey ( a , b , n ( #2:Name , #3:Fresh ) ) , n ( b , #1:Fresh ) ) ) ,
          - ( e ( seskey ( a , b , n ( #2:Name , #3:Fresh ) ) , dec ( n ( b , #1:Fresh ) ) ) ) , nil ]
```

5.3.2. Attack-state(1). Comprobación de secreto.

La especificación del attack-state(1) en Maude-NPA sería:

```
eq ATTACK-STATE(1) =
   :: r, r' ::
--- Bob's Strand
   [ nil ,-(a),
        +(e(mkey(b,s), a ; n(b,r))),
        -(e(mkey(b,s), SK ; n(b,r) ;  a)),
        +(e(SK, n(b,r'))) ,
        -(e(SK, dec(n(b,r')))) | nil ]
   | | SK inI
   | | nil
   | | nil
   | | nil
 [nonexec] .
```

En este caso indicamos que el intruso podría conocer la clave de sesión entre Alice y Bob generada por el Servidor.

En la ejecución de este ataque no se ha obtenido ninguna solución ni se ha podido finalizar la ejecución ya que el numero de estado ha sido excesivo y la herramienta no ha podido manejar el protocolo.

5.4. Wide Mouthed Frog Protocol.

El "Wide-Mouthed Frog protocol" es un protocolo de autenticación diseñado para su uso en redes inseguras (como internet, por ejemplo). Permite a los participantes comunicarse a través de una red, para probar su identidad el uno al otro a la vez que la prevención de las escuchas ilegales o ataques de repetición, y proporciona la detección de la modificación y prevención de la lectura no autorizada. El protocolo fue escrito como 'The

Wide-mouthed-frog Protocol" en [24] donde se introduce el término de "BAN logic".

El protocolo se especificaría de la siguiente manera:

(1) A → S : A,E(Kas:Ta,B,Kab)
(2) S → B : E(Kbs:Ts,A,Kab)

A genera una clave de sesión "Kab". En el mensaje 1, cuando el Servidor lo recibe, comprueba que el TimeStamp "Ta" es correcto y , si procede, reenviar la clave a Bob con su propio TimeStamp "Ts". Bob comprueba, en el mensaje 2, que el TimeStamp y que el mensaje es posterior a cualquier otro que haya recibido del Servidor. El protocolo es inseguro.

Debido a la imposibilidad de utilizar los TimeStamps en Maude-MPA hemos utilizado la versión descrita en la Sección 3.2.2 de [4].

Los participantes (Alice y Bob) comparten las claves "Kas" y "Kbs" respectivamente, con el Servidor. Cuando A y B quieren comunicarse de manera segura, A crea una nueva clave Kab, la envía al servidor, codificada bajo la clave "Kas", y el servidor la reenvía a Bob codificándola bajo "Kbs".

El protocolo se especificaría de la siguiente manera

(1) A → S : A,E(Kas:B,Kab)
(2) S → B : E(Kbs:A,Kab)
(3) A → B : A,E(Kab:M)

Los subtipos y operadores del protocolo, especificados en Maude-NPA quedarían de la siguiente manera:

```
--- Sort Information
 sorts  UName SName Name Key Nonce  Masterkey Sessionkey .
 subsort Name Nonce  Key < Msg .
 subsort Masterkey  Sessionkey < Key .
 subsort SName UName < Name .
 subsort Name < Public . --- This is quite relevant and necessary

 --- Nonce operator
 op n : Name Fresh -> Nonce [frozen] .
 op mr : Name Fresh -> Nonce [frozen] . --- Nonce, run identifier
 --- User names
 ops a b i : -> UName .
 --- Server name
 op s : ->  SName .

--- MKey
 op mkey : Name Name -> Masterkey [frozen comm] .

--- Seskey
 op seskey : Name Name Nonce -> Sessionkey [frozen] .

 ---encrypt
 op e  : Key Msg -> Msg [frozen] .
 op d : Key Msg -> Msg  [frozen] .

--- successor
```

```
op p : Msg -> Msg [frozen] .
--- Concatenation
op _;_ : Msg Msg -> Msg [frozen gather (e E)] .
```

Las propiedades algebraicas serían las siguientes:

```
fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
 protecting PROTOCOL-EXAMPLE-SYMBOLS .

 ------------------------------------------------------------
 --- Overwrite this module with the algebraic properties
 --- of your protocol
 ------------------------------------------------------------

 eq d(K:Key, e(K:Key, Z:Msg)) = Z:Msg [metadata "variant"] .
 eq e(K:Key, d(K:Key, Z:Msg)) = Z:Msg [metadata "variant"] .

endfm
```

En este protocolo existen tres roles: Servidor, Alice y Bob.

La especificación del comportamiento del Servidor es la siguiente:

```
  :: r ::
   --- Server's Strand
  [ nil | -(A ; e(mkey(A,s) , B ; SK)),
          +(e(mkey(B,s) , A  ; SK)) , nil]
```

La especificación del comportamiento de Alice es la siguiente:

```
 --- Alice's Strand.
 = :: nil ::
   [ nil | +(A ; e(mkey(A,s) , B ; seskey(A,B,n(A,r)))) ,
           +(A ; e(seskey(A,B,n(A,r)),NMA)), nil ]
```

La especificación del comportamiento de Bob es la siguiente:

```
  :: nil ::
 --- Bob's Strand.
  [ nil | -(e(mkey(B,s), A ; SK)),
          -(A ; e(SK,NMA)), nil ]
```

En este protocolo vamos a considerar dos configuraciones distintas que corresponden a las siguientes propiedades:

1. Una ejecución normal del protocolo (attack-state(0)).
2. Una ejecución donde el intruso averigua el mensaje intercambiado entre Alice y Bob (attack-state(1)).

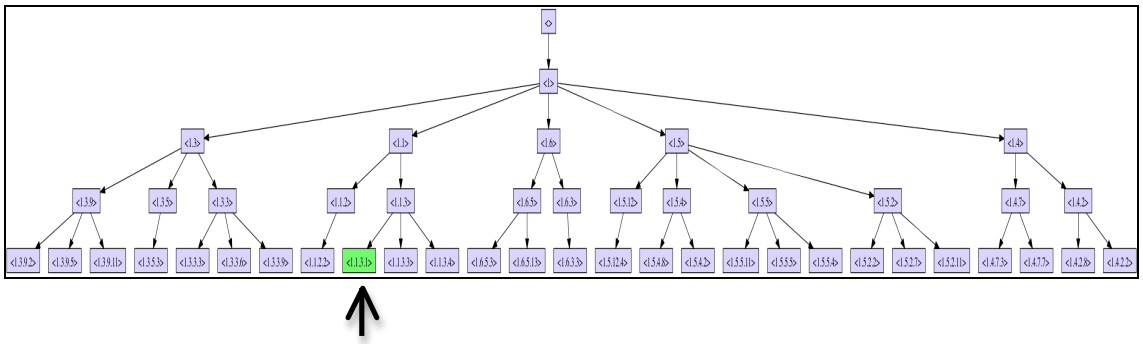
5.4.1. Attack-state(0). Ejecución normal.

Especificaríamos el attack-state(0) en Maude-NPA de la siguiente manera:

```
eq ATTACK-STATE(0) =
--- Un estado con una ejecución normal del protocolo. Añadimos el strand con el ultimo -(M)
--- y añadimos el strand que genera la clave de sesión
:: r ::
    [ nil, +(a ; e(mkey(a,s) , b ; seskey(a,b,n(a,r)))) ,
           +(a ; e(seskey(a,b,n(a,r)), NMA)) | nil ]
  &
:: nil ::
   [ nil,  -(e(mkey(b,s), a ; seskey(a,b,n(a,r)))),
           -(a ; e(seskey(a,b,n(a,r)), NMA)) | nil ]
    | | empty
    | | nil
    | | nil
    | | nil
[nonexec] .
```

Procederíamos a ejecutar el attack-state(0) mediante la herramienta gráfica de análisis.

Mostramos parte del árbol de búsqueda a continuación:



Donde podemos ver que existe una solución.

La secuencia de mensajes generados para la solución sería la siguiente:

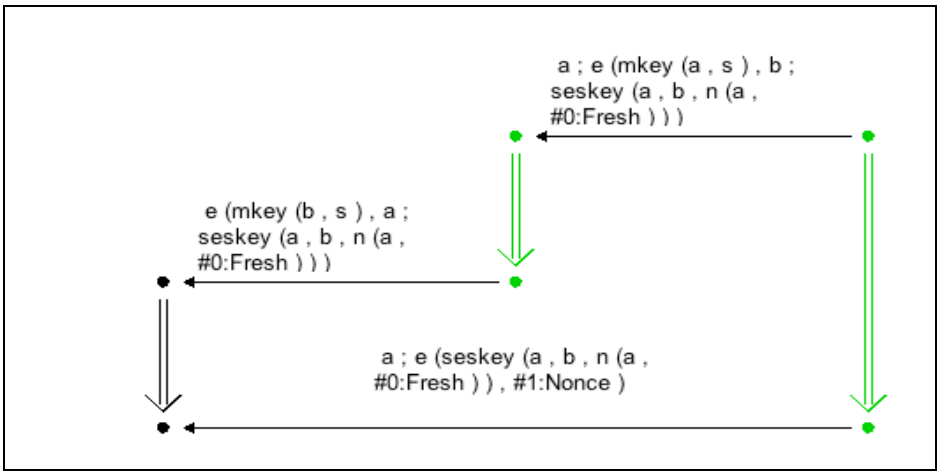
```
+ ( a ; e ( mkey ( a , s ) , b ; seskey ( a , b , n ( a , #0:Fresh ) ) ) ) ,
- ( a ; e ( mkey ( a , s ) , b ; seskey ( a , b , n ( a , #0:Fresh ) ) ) ) ,
+ ( e ( mkey ( b , s ) , a ; seskey ( a , b , n ( a , #0:Fresh ) ) ) ) ,
+ ( a ; e ( seskey ( a , b , n ( a , #0:Fresh ) ) , #1:Nonce ) ) ,
- ( e ( mkey ( b , s ) , a ; seskey ( a , b , n ( a , #0:Fresh ) ) ) ) ,
- ( a ; e ( seskey ( a , b , n ( a , #0:Fresh ) ) , #1:Nonce ) )
```

Podríamos expresarlo en notación informal de la siguiente manera:

(1) Z(A) → Z(S) : A,E(Kas:B,Kab)
(2) Z(S) → B : E(Kbs:A,Kab)
(3) Z(A) → B : A,E(Kab:M)

Podemos observar que el protocolo tampoco es seguro ante un ataque de autenticación.

La visualización gráfica de la solución obtenida sería la siguiente:



Podemos ver que el protocolo no es seguro.

El strand de Bob generado por la herramienta gráfica sería el siguiente:

```
:: nil ::
[ nil | - ( e ( mkey ( b , s ) , a ; seskey ( a , b , n ( a , #0:Fresh ) ) ) ) ,
        - ( a ; e ( seskey ( a , b , n ( a , #0:Fresh ) ) , #1:Nonce ) ) , nil ] &
```

El strand del Servidor generado por la herramienta gráfica sería el siguiente:

```
:: nil ::
   [ nil | - ( a ; e ( mkey ( a , s ) , b ; seskey ( a , b , n ( a , #0:Fresh ) ) ) ) ,
           + ( e ( mkey ( b , s ) , a ; seskey ( a , b , n ( a , #0:Fresh ) ) ) ) , nil ] &
```

El strand de Alice generado por la herramienta gráfica sería el siguiente:

```
:: #0:Fresh ::
  [ nil | + ( a ; e ( mkey ( a , s ) , b ; seskey ( a , b , n ( a , #0:Fresh ) ) ) ) ,
          + ( a ; e ( seskey ( a , b , n ( a , #0:Fresh ) ) , #1:Nonce ) ) , nil ]
```

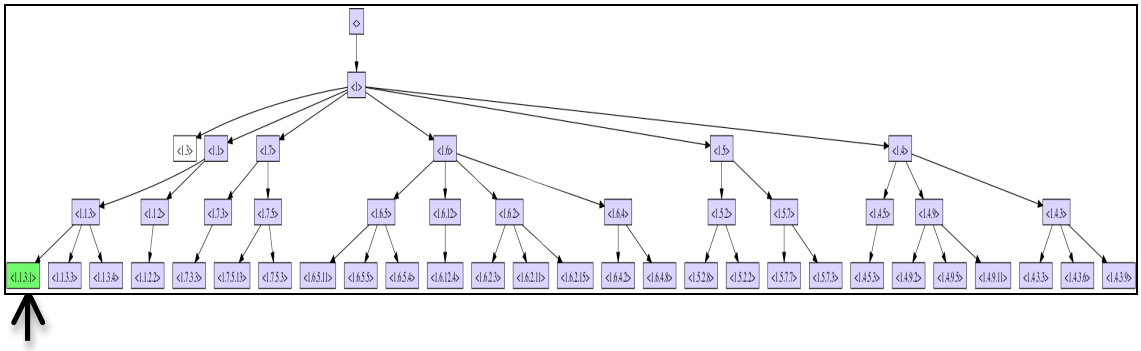
5.4.2. Attack-state(1). Comprobación de secreto.

El attack-state(1) sería el siguiente:

```
eq ATTACK-STATE(1) =
--- Un estado con una ejecución normal del protocolo. Añadimos el strand con el ultimo -(M)
--- y añadimos el strand que genera la clave de sesión
:: r ::
    [ nil, +(a ; e(mkey(a,s) , b ; seskey(a,b,n(a,r)))) ,
           +(a ; e(seskey(a,b,n(a,r)), NMA)) | nil ]
  &
:: nil ::
   [ nil,  -(e(mkey(b,s), a ; seskey(a,b,n(a,r)))),
           -(a ; e(seskey(a,b,n(a,r)), NMA)) | nil ]
    | | NMA inl
    | | nil
    | | nil
    | | nil
[nonexec] .
```

Aquí indicamos que el intruso pudiera conocer el mensaje intercambiado entre Alice y Bob.

Parte del árbol de búsqueda que se obtiene usando la herramienta gráfica sería el siguiente:



Donde podemos ver que existe una solución, por lo tanto el protocolo no es seguro ante un ataque de obtención de la clave de sesión.

La secuencia de mensajes generados para la solución sería la siguiente:

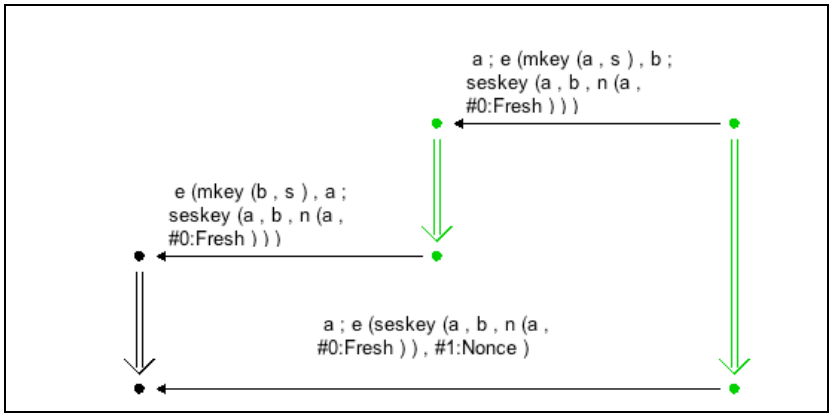
```
+ ( a ; e ( mkey ( a , s ) , b ; seskey ( a , b , n ( a , #0:Fresh ) ) ) ) ,
- ( a ; e ( mkey ( a , s ) , b ; seskey ( a , b , n ( a , #0:Fresh ) ) ) ) ,
+ ( e ( mkey ( b , s ) , a ; seskey ( a , b , n ( a , #0:Fresh ) ) ) ) ,
generatedByIntruder ( #1:Nonce ) ,
+ ( a ; e ( seskey ( a , b , n ( a , #0:Fresh ) ) , #1:Nonce ) ) ,
- ( e ( mkey ( b , s ) , a ; seskey ( a , b , n ( a , #0:Fresh ) ) ) ) ,
- ( a ; e ( seskey ( a , b , n ( a , #0:Fresh ) ) , #1:Nonce ) )
```

Podríamos expresarlo en notación informal de la siguiente manera:

1. Z(A) → Z(S) : A,E(Kas:B,Kab)
2. Z(S) → B : E(Kbs:A,Kab)
3. Z(A) → B : A,E(Kab:M')

Donde M' es el mensaje generado por el intruso.

La visualización gráfica de la solución obtenida sería la siguiente:



5.5. Yahalom protocol.

Yahalom es un protocolo de autenticación y compartición de clave segura diseñado para su uso en una red insegura como Internet. Yahalom utiliza un mediador (Servidor) de confianza para distribuir una clave compartida entre los participantes. Este protocolo se puede considerar como una versión mejorada del protocolo Wide Mouth Frog (con una protección adicional contra el ataque man-in-the-middle), pero menos seguro que Needham-Schroeder. La descripción informal del protocolo proporcionada en la sección 6.3.6 de [1] , pág. 49, es la siguiente:

(1) A → B : A,Na
(2) B → S : B,E(Kbs:A,Na,Nb)
(3) S → A : E(Kas:B,Kab,Na,Nb),E(Kbs:A,Kab)
(4) A → B : E(Kbs:A,Kab),E(Kab:Nb)

En el mensaje 1, Alice envía un mensaje a Bob para iniciar la comunicación, donde le envía además su Nonce "Na". A continuación, en el mensaje 2, Bob envía un mensaje al Servidor encriptado bajo la clave "Kbs". Después, en el mensaje 3, el Servidor envía a Alice un mensaje que contiene la clave de sesión generada "Kab" y un mensaje para ser enviado a Bob. Finalmente, en el mensaje 4, Alice envía el mensaje a Bob y verifica que el nonce "Na" no ha cambiado. También Bob, cuando recibe el mensaje, verifica que el nonce "Nb" no ha cambiado.

Los subtipos y operadores del protocolo, especificados en Maude-NPA quedarían de la siguiente manera:

```
--- Sort Information
 subsort Name Nonce  Key < Msg .
 subsort Masterkey  Sessionkey < Key .
 subsort SName UName < Name .
 subsort Name < Public .

--- Nonce operator
 op n : Name Fresh -> Nonce [frozen] .

 --- User names
 ops a b i : -> UName .

 --- Server name
 op s : ->  SName .

--- MKey
 op mkey : Name Name -> Masterkey [frozen comm] .

--- Seskey
 op seskey : Name Name Nonce -> Sessionkey [frozen] .

---encrypt
 op e  : Key Msg -> Msg [frozen] .
 op d : Key Msg -> Msg  [frozen] .

--- Concatenation
 op _;_ : Msg Msg -> Msg [frozen gather (e E)] .
```

Las propiedades algebraicas serían las siguientes:

```
fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
  protecting PROTOCOL-EXAMPLE-SYMBOLS .

  ------------------------------------------------------------
  --- Overwrite this module with the algebraic properties
  --- of your protocol
  ------------------------------------------------------------

 eq d(K:Key, e(K:Key, Z:Msg)) = Z:Msg [metadata "variant"] .
 eq e(K:Key, d(K:Key, Z:Msg)) = Z:Msg [metadata "variant"] .

endfm
```

En este protocolo existen tres roles: Servidor, Alice y Bob.

La especificación del comportamiento del Servidor es la siguiente:

```
  = :: r ::
--- Alice's Strand
   [ nil |  +(A ; n(A,r)),
            -(e(mkey(A,s) , B ; SK ; n(A,r) ; NB) ; MB),
            +(MB ; e(SK , NB)) , nil ]
```

La especificación del comportamiento de Bob es la siguiente:

```
    :: r ::
---Bob's Strand
   [ nil |  -(A ; NA),
            +(B ; e(mkey(B,s) , A ; NA ; n(B,r))),
             -(e(mkey(B,s) , A ; SK) ; e(SK , n(B,r))) , nil ]
```

La especificación del comportamiento de Alice es la siguiente:

```
    :: r ::
---Server's Strand
   [ nil |  -(B ; e(mkey(B,s) , A ; NA ; NB)),
            +(e(mkey(A,s) , B ; seskey(A , B , n(s,r)) ; NA ; NB) ;
              e(mkey(B,s) , A ; seskey(A , B , n(s,r)))) , nil ]
```

En este protocolo vamos a considerar tres configuraciones distintas que corresponden a las siguientes propiedades:

1. Una ejecución normal del protocolo (attack-state(0)).
2. Una ejecución donde el intruso averigua la clave de Alice y Bob generada por el servidor (attack-state(1)).
3. Una ejecución donde Bob ha completado el protocolo creyendo que habla con Alice pero no es así (attack-state(2)).

5.5.1. Attack-state(0). Ejecución normal.

Especificaríamos el attack-state(0) en Maude-NPA de la siguiente manera:

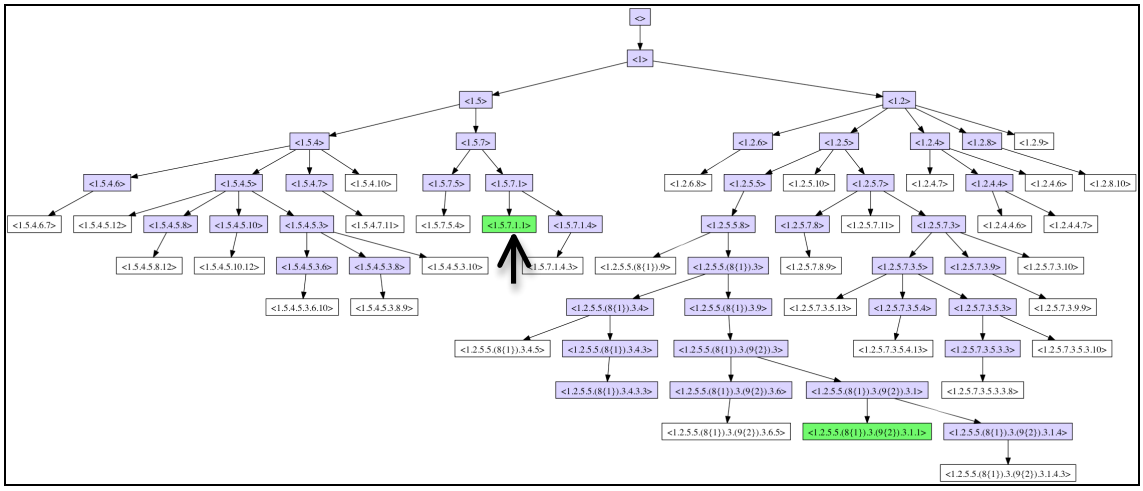
```
eq ATTACK-STATE(0) =
   :: r ::
--- Un estado con una ejecución normal del protocolo. añadimos el strand con el último -(M)
   [ nil ,  -(a ; NA),
            +(b ; e(mkey(b,s) , a ; NA ; n(b,r))),
             -(e(mkey(b,s) , a ; SK) ; e(SK , n(b,r)))  |  nil ]
    | | empty
    | | nil
    | | nil
    | | nil
[nonexec] .
```

Incluimos sólo el strand de Bob, sin requerir información sobre que conoce el intruso y la herramienta rellenará el resto.

Como podemos ver, la ejecución del attack-state(0) es una ejecución normal del protocolo.

Procederíamos a ejecutar el attack-state(0) mediante la herramienta gráfica de análisis.

Mostramos el árbol de búsqueda a continuación:



Donde podemos ver que existe una solución y por lo tanto una ejecución normal del protocolo.

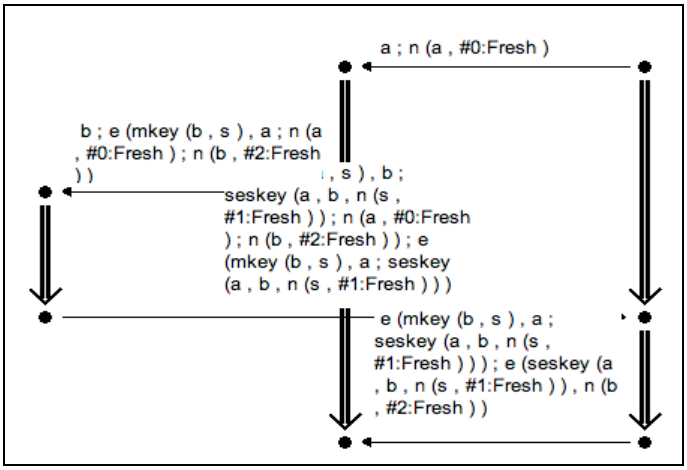
La secuencia de mensajes generados para la solución sería la siguiente:

```
+ ( a ; n ( a , #0:Fresh ) ) ,
- ( a ; n ( a , #0:Fresh ) ) ,
+ ( b ; e ( mkey ( b , s ) , a ; n ( a , #0:Fresh ) ; n ( b , #2:Fresh ) ) ) ,
- ( b ; e ( mkey ( b , s ) , a ; n ( a , #0:Fresh ) ; n ( b , #2:Fresh ) ) ) ,
+ ( e ( mkey ( a , s ) , b ; seskey ( a , b , n ( s , #1:Fresh ) ) ;
    n ( a , #0:Fresh ) ; n ( b , #2:Fresh ) ) ;
    e ( mkey ( b , s ) , a ; seskey ( a , b , n ( s , #1:Fresh ) ) ) ) ,
- ( e ( mkey ( a , s ) , b ; seskey ( a , b , n ( s , #1:Fresh ) ) ;
    n ( a , #0:Fresh ) ; n ( b , #2:Fresh ) ) ;
    e ( mkey ( b , s ) , a ; seskey ( a , b , n ( s , #1:Fresh ) ) ) ) ,
+ ( e ( mkey ( b , s ) , a ; seskey ( a , b , n ( s , #1:Fresh ) ) ) ;
    e ( seskey ( a , b , n ( s , #1:Fresh ) ) , n ( b , #2:Fresh ) ) ) ,
- ( e ( mkey ( b , s ) , a ; seskey ( a , b , n ( s , #1:Fresh ) ) ) ;
    e ( seskey ( a , b , n ( s , #1:Fresh ) ) , n ( b , #2:Fresh ) ) )
```

Podríamos expresarlo en notación informal de la siguiente manera:

(1) A → B : A,Na
(2) B → S : B,E(Kbs:A,Na,Nb)
(3) S → A : E(Kas:B,Kab,Na,Nb),E(Kbs:A,Kab)
(4) A → B : E(Kbs:A,Kab),E(Kab:Nb)

La visualización gráfica de la solución obtenida sería la siguiente:



Donde podemos observar que se produce una ejecución normal del protocolo.

El strand de Alice generado por la herramienta gráfica sería el siguiente:

```
:: #0:Fresh ::
--- Alice's Strand
[ nil | + ( a ; n ( a , #0:Fresh ) ) ,
        - ( e ( mkey ( a , s ) , b ; seskey ( a , b , n ( s , #1:Fresh ) ) ;
            n ( a , #0:Fresh ) ; n ( b , #2:Fresh ) ) ;
            e ( mkey ( b , s ) , a ; seskey ( a , b , n ( s , #1:Fresh ) ) ) ) ,
        + ( e ( mkey ( b , s ) , a ; seskey ( a , b , n ( s , #1:Fresh ) ) ) ;
            e ( seskey ( a , b , n ( s , #1:Fresh ) ) , n ( b , #2:Fresh ) ) ) , nil ]
```

El strand de Bob generado por la herramienta gráfica sería el siguiente:

```
:: #2:Fresh ::
--- Bob's Strand
[ nil | - ( a ; n ( a , #0:Fresh ) ) ,
      + ( b ; e ( mkey ( b , s ) , a ; n ( a , #0:Fresh ) ;
          n ( b , #2:Fresh ) ) ) ,
      - ( e ( mkey ( b , s ) , a ; seskey ( a , b , n ( s , #1:Fresh ) ) ) ;
          e ( seskey ( a , b , n ( s , #1:Fresh ) ) , n ( b , #2:Fresh ) ) ) , nil ] &
```

El strand del Servidor generado por la herramienta gráfica sería el siguiente:

```
:: #1:Fresh ::
--- Server's Strand
[ nil | - ( b ; e ( mkey ( b , s ) , a ; n ( a , #0:Fresh ) ;
            n ( b , #2:Fresh ) ) ) ,
        + ( e ( mkey ( a , s ) , b ;
            seskey ( a , b , n ( s , #1:Fresh ) ) ;
            n ( a , #0:Fresh ) ; n ( b , #2:Fresh ) ) ;
            e ( mkey ( b , s ) , a ; seskey ( a , b , n ( s , #1:Fresh ) ) ) ) , nil ] &
```

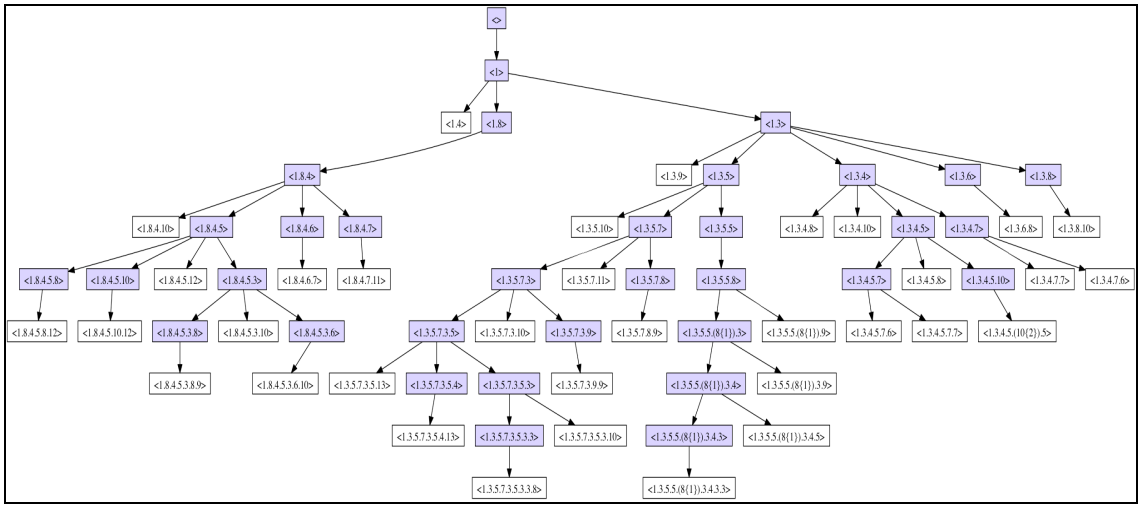
5.5.2. Attack-state(1). Comprobación de secreto.

El attack-state(1) sería el siguiente:

```
eq ATTACK-STATE(1) =
:: r ::
--- Un estado con un secreto por descubrir
     [ nil ,  -(a ; NA),
              +(b ; e(mkey(b,s) , a ; NA ; n(b,r))),
               -(e(mkey(b,s) , a ; SK) ; e(SK , n(b,r)))  |  nil ]
    | | SK inI
    | | nil
    | | nil
    | | nil
[nonexec] .
```

Aquí indicamos que el intruso pudiera conocer la clave de sesión entre Alice y Bob generada por el Servidor.

El árbol de búsqueda que se obtiene usando la herramienta gráfica sería el siguiente.



Donde podemos ver que no existe una solución. Por lo tanto es seguro ante un ataque de obtención de la clave de sesión.

5.5.3. Attack-state(2). Comprobación de autenticación.

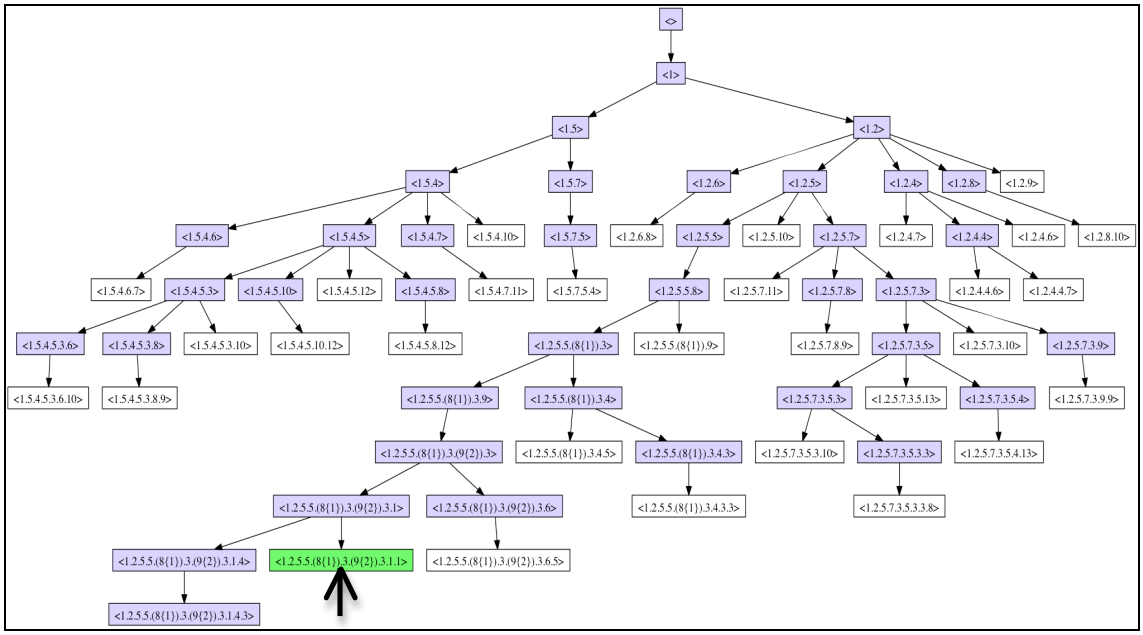
El attack-state(2) sería el siguiente:

```
eq ATTACK-STATE(2) =
--- Un estado con una ejecucion normal pero con patron de autenticacion.
:: r ::
   [ nil ,  -(a ; NA),
            +(b ; e(mkey(b,s) , a ; NA ; n(b,r))),
            -(e(mkey(b,s) , a ; SK) ; e(SK , n(b,r)))  | nil ]
| | empty
| | nil
| | nil
| | never
*** Pattern for authentication
    (:: R:FreshSet ::
    [ nil |    +(a ; NA),
               -(MA ; e(mkey(b,s) , a ; SK)),
               +(e(mkey(b,s) , a ; SK) ; e(SK , n(b,r))) , nil ]
     & S:StrandSet | | K:IntruderKnowledge)

[nonexec] .
```

En este caso se utiliza un patrón de autenticación que sería el strand de ejecución de Alice.

Mostramos parte del árbol de búsqueda a continuación.



Donde podemos ver que existe una solución, por lo tanto no es seguro ante un ataque de autenticación.

La secuencia de mensajes generados para la solución sería la siguiente:

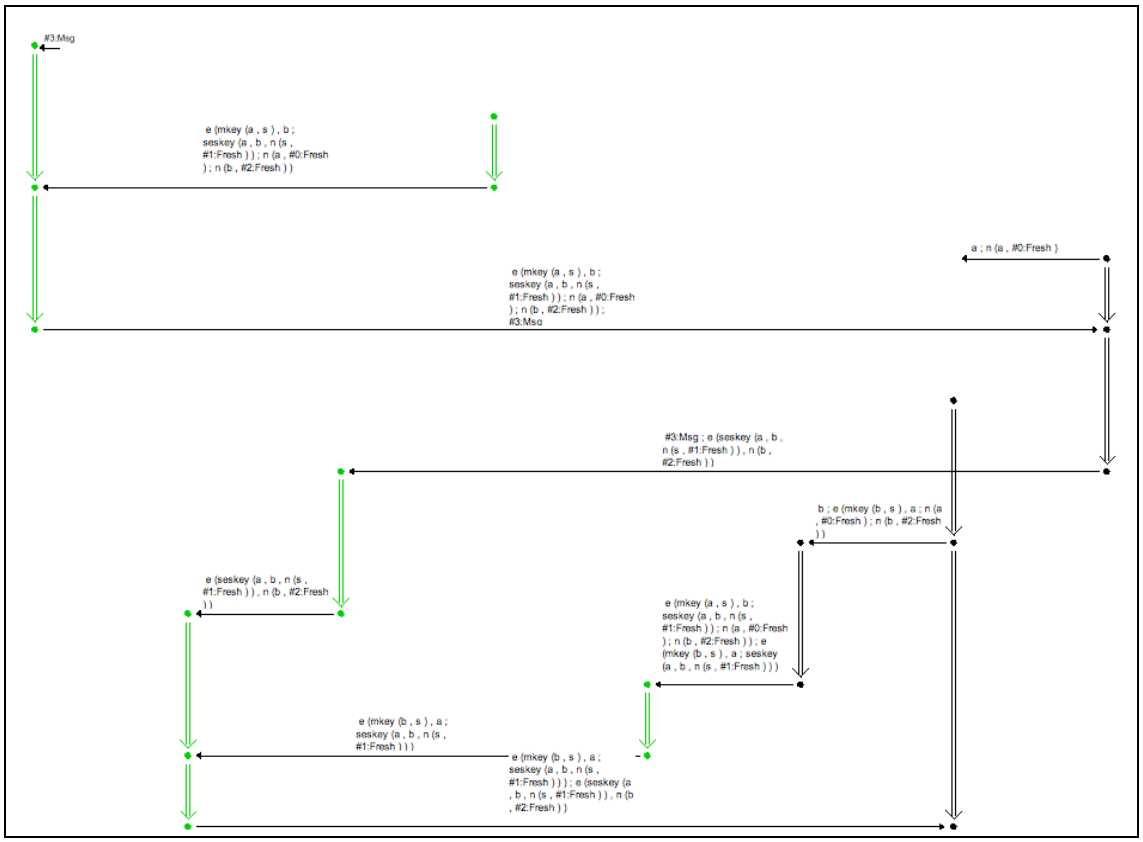
```
+ ( a ; n ( a , #0:Fresh ) ) ,
- ( a ; n ( a , #0:Fresh ) ) ,
+ ( b ; e ( mkey ( b , s ) , a ; n ( a , #0:Fresh ) ; n ( b , #2:Fresh ) ) ) ,
- ( b ; e ( mkey ( b , s ) , a ; n ( a , #0:Fresh ) ; n ( b , #2:Fresh ) ) ) ,
+ ( e ( mkey ( a , s ) , b ; seskey ( a , b , n ( s , #1:Fresh ) ) ;
    n ( a , #0:Fresh ) ; n ( b , #2:Fresh ) ) ;
    e ( mkey ( b , s ) , a ; seskey ( a , b , n ( s , #1:Fresh ) ) ) ) ,
- ( e ( mkey ( a , s ) , b ; seskey ( a , b , n ( s , #1:Fresh ) ) ;
   n ( a , #0:Fresh ) ; n ( b , #2:Fresh ) ) ;
   e ( mkey ( b , s ) , a ; seskey ( a , b , n ( s , #1:Fresh ) ) ) ) ,
+ ( e ( mkey ( a , s ) , b ; seskey ( a , b , n ( s , #1:Fresh ) ) ;
    n ( a , #0:Fresh ) ; n ( b , #2:Fresh ) ) ) ,
generatedByIntruder ( #3:Msg ) ,
- ( #3:Msg ) ,
- ( e ( mkey ( a , s ) , b ; seskey ( a , b , n ( s , #1:Fresh ) ) ;
   n ( a , #0:Fresh ) ; n ( b , #2:Fresh ) ) ) ,
+ ( e ( mkey ( a , s ) , b ; seskey ( a , b , n ( s , #1:Fresh ) ) ;
   n ( a , #0:Fresh ) ; n ( b , #2:Fresh ) ) ; #3:Msg ) ,
- ( e ( mkey ( a , s ) , b ; seskey ( a , b , n ( s , #1:Fresh ) ) ;
   n ( a , #0:Fresh ) ; n ( b , #2:Fresh ) ) ; #3:Msg ) ,
+ ( #3:Msg ; e ( seskey ( a , b , n ( s , #1:Fresh ) ) , n ( b , #2:Fresh ) ) ) ,
- ( e ( mkey ( a , s ) , b ; seskey ( a , b , n ( s , #1:Fresh ) ) ;
   n ( a , #0:Fresh ) ; n ( b , #2:Fresh ) ) ;
   e ( mkey ( b , s ) , a ; seskey ( a , b , n ( s , #1:Fresh ) ) ) ) ,
+ ( e ( mkey ( b , s ) , a ; seskey ( a , b , n ( s , #1:Fresh ) ) ) ) ,
- ( #3:Msg ; e ( seskey ( a , b , n ( s , #1:Fresh ) ) , n ( b , #2:Fresh ) ) ) ,
+ ( e ( seskey ( a , b , n ( s , #1:Fresh ) ) , n ( b , #2:Fresh ) ) ) ,
- ( e ( seskey ( a , b , n ( s , #1:Fresh ) ) , n ( b , #2:Fresh ) ) ) ,
- ( e ( mkey ( b , s ) , a ; seskey ( a , b , n ( s , #1:Fresh ) ) ) ) ,
+ ( e ( mkey ( b , s ) , a ; seskey ( a , b , n ( s , #1:Fresh ) ) ) ;
   e ( seskey ( a , b , n ( s , #1:Fresh ) ) , n ( b , #2:Fresh ) ) ) ,
- ( e ( mkey ( b , s ) , a ; seskey ( a , b , n ( s , #1:Fresh ) ) ) ;
   e ( seskey ( a , b , n ( s , #1:Fresh ) ) , n ( b , #2:Fresh ) ) )
```

Podríamos expresarlo en notación informal de la siguiente manera:

(1)	A → B	: A, Na
(2)	Z → A	: E(Kas : B, Kab, Na ,Nb) ,E(Kbs : Z, Kab)
(3)	A→Z(B)	: E(Kbs : Z, Kab), E(Kab : Nb)
(4)	B→ S	: B, E(Kbs : A, Na ,Nb)
(5)	S→ Z(A)	: E(Kas : B, Kab, Na ,Nb) ,E(Kbs : A, Kab)
(6)	Z(A)→ B	: E(Kbs : A, Kab), E(Kab : Nb)

En la visualización gráfica, donde se muestra "#3:Msg" lo hemos interpretado como "E(Kbs:Z,Kab)".

La visualización gráfica de la solución obtenida sería la siguiente:



5.6. Carlsen's Secret Key Initiator Protocol.

La descripción informal del protocolo proporcionada en la sección 6.3.7 de [1] , pág. 50, sería la siguiente:

(1) A → B : A,Na
(2) B → S : A,Na,B,Nb
(3) S → B : E(Kbs:Kab,Nb,A),E(Kas:Na,B,Kab)
(4) B → A : E(Kas:Na,B,Kab),E(Kab:Na),N'b
(5) A → B : E(Kab:N'b)

Los tres primeros strands y la primera mitad del cuarto strand son exactamente como la simplificación que hacen Abadi y Needham del protocolo de Otway-Rees (sección 5.2 de éste documento). Lo que sigue son los reconocimientos de Alice y Bob de que se ha recibido la clave de sesión "Kab". Bob reconoce a Alice mediante el envío de E(Kab:Na) y Alice reconoce a Bob enviando E(Kab:N'b).

Los subtipos y operadores del protocolo, especificados en Maude-NPA quedarían de la siguiente manera:

```
--- Sort Information
 subsort Name Nonce  Key < Msg .
 subsort Masterkey  Sessionkey < Key .
 subsort SName UName < Name .
 subsort Name < Public .
--- Nonce operator
 op n : Name Fresh -> Nonce [frozen] .
 op mr : Name Fresh -> Nonce [frozen] . --- Nonce, run identifier

 --- User names
 ops a b i : -> UName .

 --- Server name
 op s : ->  SName .

--- MKey
 op mkey : Name Name -> Masterkey [frozen comm] .

--- Seskey
 op seskey : Name Name Nonce -> Sessionkey [frozen] .

 ---encrypt
 op e  : Key Msg -> Msg [frozen] .
 op d : Key Msg -> Msg  [frozen] .

--- successor

 op p : Msg -> Msg [frozen] .

--- Concatenation
 op _;_ : Msg Msg -> Msg [frozen gather (e E)] .
```

Las propiedades algebraicas serían las siguientes:

```
fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
 protecting PROTOCOL-EXAMPLE-SYMBOLS .

 ------------------------------------------------------------
 --- Overwrite this module with the algebraic properties
 --- of your protocol
 ------------------------------------------------------------

 eq d(K:Key, e(K:Key, Z:Msg)) = Z:Msg [metadata "variant"] .
 eq e(K:Key, d(K:Key, Z:Msg)) = Z:Msg [metadata "variant"] .


endfm
```

En este protocolo existen tres roles: Alice, Bob y Servidor.

La especificación del comportamiento del Servidor es la siguiente:

```
  :: r ::
   --- Server's Strand
  [ nil |  -(A ; NA ; B ; NB),
           +(e(mkey(B,s) , seskey(A , B , n(S,r)) ; NB ; A) ;
              e(mkey(A,s) , NA ; B ; seskey(A , B , n(S,r)))), nil]
```

La especificación del comportamiento de Alice es la siguiente:

```
--- Alice's Strand.
= :: r ::
  [ nil | +(A ; n(A,r)),
          -(e(mkey(A,s) , n(A,r) ; B ; SK) ; e(SK , n(A,r)) ; NB1),
          +(e(SK , NB1)), nil ]
```

La especificación del comportamiento de Bob es la siguiente:

```
 :: r , r1 ::
--- Bob's Strand.
  [ nil | -(A ; NA),
          +(A ; NA ; B ; n(B,r)),
          -(e(mkey(B,s) , SK ; n(B,r) ; A) ; MA),
          +(MA ; e(SK , NA) ; n(B,r1)),
          -(e(SK , n(B,r1))), nil ]
```

En este protocolo vamos a considerar tres configuraciones distintas que corresponden a las siguientes propiedades:

1. Una ejecución normal del protocolo (attack-state(0)).
2. Una ejecución donde el intruso averigua la clave de Alice y Bob generada por el servidor (attack-state(1)).
3. Una ejecución donde Bob ha completado el protocolo creyendo que habla con Alice pero no es así (attack-state(2)).

5.6.1. Attack-state(0). Ejecución normal.

Especificaríamos el attack-state(0) en Maude-NPA de la siguiente manera:

```
eq ATTACK-STATE(0) =
:: r,r1 ::
--- Un estado con una ejecución normal del protocolo. añadimos el strand con el último -(M)
---Bob's Strand
   [ nil ,  -(a ; NA),
            +(a ; NA ; b ; n(b,r)),
            -(e(mkey(b,s) , SK ; n(b,r) ; a) ; MA),
            +(MA ; e(SK , NA) ; n(b,r1)),
            -(e(SK , n(b,r1)))  | nil ]
    | | empty
    | | nil
    | | nil
    | | nil
[nonexec] .
```

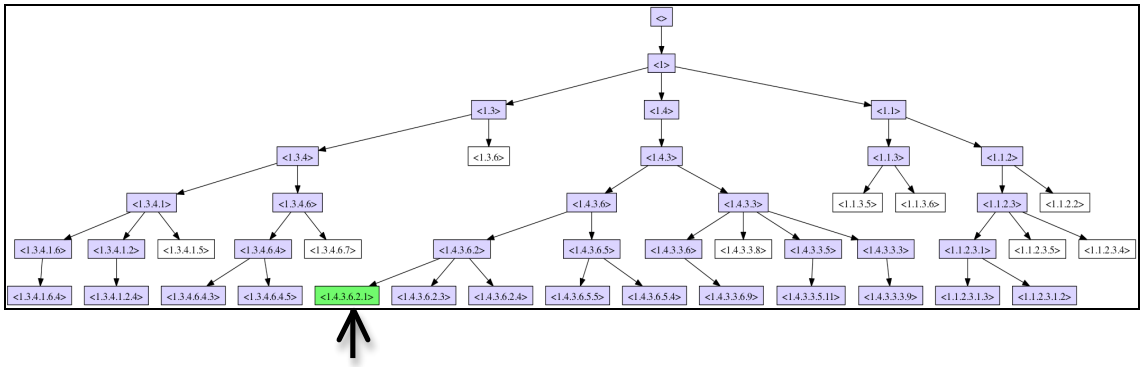
Incluimos sólo el strand de Bob, sin requerir información sobre que conoce el intruso y la herramienta rellenará el resto.

Procederíamos a ejecutar el attack-state(0) mediante la herramienta gráfica de análisis.

Como podemos ver, la ejecución del attack-state(0) es una ejecución normal del protocolo.

En el tercer strand reemplazamos “e(mkey(A,s) , n(A,r) ; B ; SK” por MA.

Mostramos parte del árbol de búsqueda a continuación.



Donde podemos ver que existe una solución, y por lo tanto una ejecución normal del protocolo.

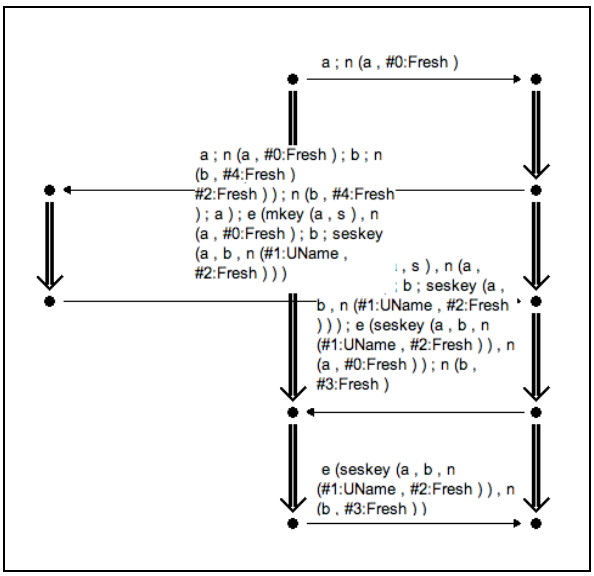
La secuencia de mensajes generados para la solución sería la siguiente:

```
+ ( a ; n ( a , #0:Fresh ) ) ,
 - ( a ; n ( a , #0:Fresh ) ) ,
+ ( a ; n ( a , #0:Fresh ) ; b ; n ( b , #4:Fresh ) ) ,
 - ( a ; n ( a , #0:Fresh ) ; b ; n ( b , #4:Fresh ) ) ,
+ ( e ( mkey ( b , s ) , seskey ( a , b , n ( #1:UName , #2:Fresh ) ) ;
    n ( b , #4:Fresh ) ; a ) ;
    e ( mkey ( a , s ) , n ( a , #0:Fresh ) ; b ;
    seskey ( a , b , n ( #1:UName , #2:Fresh ) ) ) ) ,
- ( e ( mkey ( b , s ) , seskey ( a , b , n ( #1:UName , #2:Fresh ) ) ;
   n ( b , #4:Fresh ) ; a ) ;
   e ( mkey ( a , s ) , n ( a , #0:Fresh ) ; b ;
   seskey ( a , b , n ( #1:UName , #2:Fresh ) ) ) ) ,
+ ( e ( mkey ( a , s ) , n ( a , #0:Fresh ) ; b ;
    seskey ( a , b , n ( #1:UName , #2:Fresh ) ) ) ;
    e ( seskey ( a , b , n ( #1:UName , #2:Fresh ) ) , n ( a , #0:Fresh ) ) ;
    n ( b , #3:Fresh ) ) ,
- ( e ( mkey ( a , s ) , n ( a , #0:Fresh ) ; b ;
    seskey ( a , b , n ( #1:UName , #2:Fresh ) ) ) ;
    e ( seskey ( a , b , n ( #1:UName , #2:Fresh ) ) , n ( a , #0:Fresh ) ) ;
    n ( b , #3:Fresh ) ) ,
+ ( e ( seskey ( a , b , n ( #1:UName , #2:Fresh ) ) , n ( b , #3:Fresh ) ) ) ,
 - ( e ( seskey ( a , b , n ( #1:UName , #2:Fresh ) ) , n ( b , #3:Fresh ) ) )
```

Esta secuencia en notación informal sería de la siguiente manera:

(1) A → B : A,Na
(2) B → S : A,Na,B,Nb
(3) S → B : E(Kbs:Kab,Nb,A),E(Kas:Na,B,Kab)
(4) B → A : E(Kas:Na,B,Kab),E(Kab:Na),N'b
(5) A → B : E(Kab:N'b)

La visualización gráfica de la solución obtenida sería la siguiente:



Donde podemos ver que se produce una ejecución normal.

5.6.2. Attack-state(1). Comprobación de secreto.

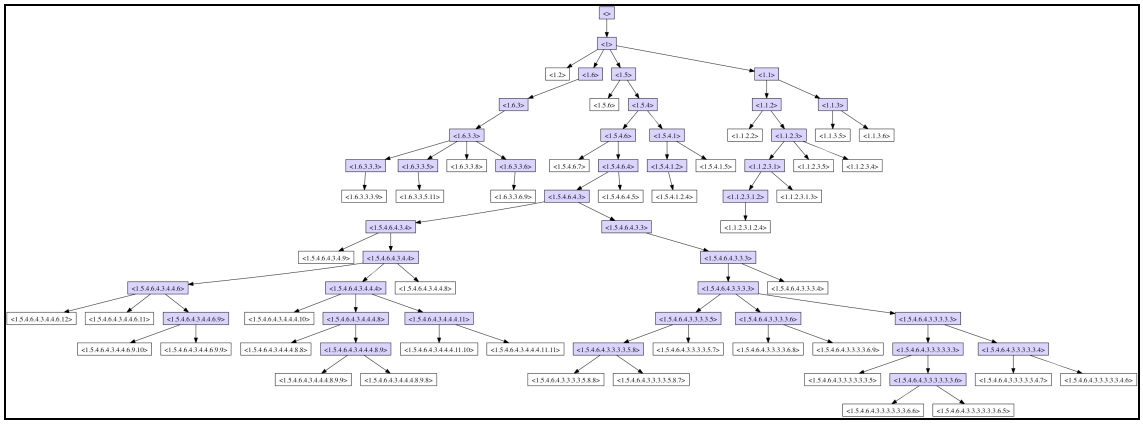
La especificación del attack-state(1) sería la siguiente:

```
eq ATTACK-STATE(1) =
:: r,r1 ::
--- Un estado con un secreto por descubrir
---Bob's Strand
   [ nil ,    -(a ; NA),
              +(a ; NA ; b ; n(b,r)),
               -(e(mkey(b,s) , SK ; n(b,r) ; a) ; MA),
              +(MA ; e(SK , NA) ; n(b,r1)),
               -(e(SK , n(b,r1))) | nil ]
    | | SK inI
    | | nil
    | | nil
    | | nil
[nonexec] .
```

En este caso indicamos que el intruso podría conocer la clave de sesión entre Alice y Bob generada por el Servidor.

Procederíamos a ejecutar el attack-state(1) mediante la herramienta gráfica de análisis.

El árbol de búsqueda que se obtiene sería el siguiente.



Donde observamos que no existe una solución. Por lo tanto es seguro ante un ataque de obtención de la clave de sesión.

5.6.3. Attack-state(1). Comprobación de autenticación.

La especificación del attack-state(2) sería la siguiente:

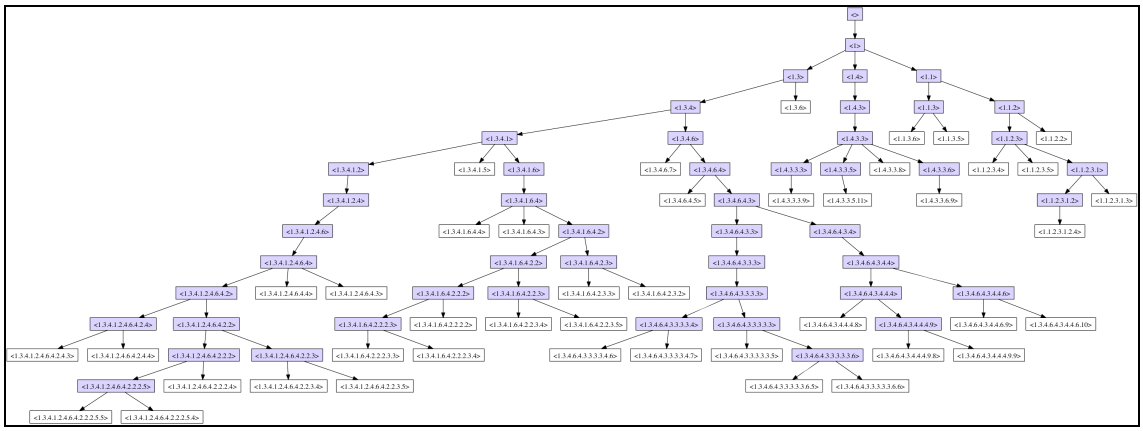
```
eq ATTACK-STATE(2) =
--- Un estado con una ejecución normal pero con patrón de autenticación.
---Bob's Strand
:: r,r1 ::
   [ nil ,    -(a ; NA),
              +(a ; NA ; b ; n(b,r)),
               -(e(mkey(b,s) , SK ; n(b,r) ; a) ; MA),
              +(MA ; e(SK , NA) ; n(b,r1)),
               -(e(SK , n(b,r1))) | nil ]
| | empty
| | nil
| | nil
| | never
 *** Pattern for authentication
 --- Alice's Strand
    (:: R:FreshSet ::
    [ nil |    +(a ; NA),
               -(e(mkey(a,s) , NA ; b ; SK) ; e(SK , NA) ; n(b,r1)),
               +(e(SK , n(b,r1))), nil ]
     & S:StrandSet | | K:IntruderKnowledge)

[nonexec] .
```

En este caso usaríamos el strand de Alice como patrón de autenticación.

Procederíamos a ejecutar el attack-state(2) mediante la herramienta gráfica de análisis.

El árbol de búsqueda que se obtiene sería el siguiente:



Donde observamos que no existe una solución. Por lo tanto el protocolo es seguro ante un ataque de autenticación.

5.7. ISO 5-pass Authentication Protocol.

En [3] encontramos una especificación de este protocolo ligeramente distinta a la que se especifica en el Clark-Jacob.

El protocolo original, especificado en [1] se ha modificado para incluir variables de aviso, una vez eliminados los mensajes de carga de extraños, y la corrección de lo que parecen ser algunos errores (por ejemplo, Alice no envía a Bob nunca su nombre, así que Bob no tiene idea de quién está contactando con él).

Este protocolo es similar al "Carlsen's Secret Key Initiator Protocol", de la sección 5.6. La única diferencia es que incluye el Nonce de B (Nb) en el mensaje 4 en "E(Kab:Nb,Na)", y que incluye el Nonce de Alice "Na" en el mensaje 5 en "E(Kab:Na,Nb)".

Las descripciones informales de las diferentes versiones del protocolo son las siguientes.

La descripción informal del protocolo proporcionada en [3] es la siguiente:

(1) A -> B : A, Na
(2) B -> S : A, Na, B, Nb'
(3) S -> B : E(Kbs:Nb',Kab,A), E(Kas:Na,Kab,B)
(4) B -> A : E(Kas:Na,Kab,B), E(Kab:Nb,Na)
(5) A -> B : E(Kab:Na,Nb)

La descripción informal del protocolo proporcionada en la sección 6.3.9 de [1], pág. 50, es la siguiente:

```
(1)  A -> B : Ra,Text1
(2)  B -> S  : R'b,Ra,A,Text2
(3)  S -> B  : Text5,E(Kbs:R'b,Kab,A,Text4),E(Kas:Ra,Kab.B,Text3)
(4)  B -> A : Text7,E(Kas:Ra,Kab.B,Text3),E(Kab:rb,Ra,text6)
(5)  A -> B : text9,E(Kab:Ra,rb,Text8)
```

Nosotros vamos a usar la primera de las dos descripciones.

Los subtipos y operadores de la versión del protocolo que hemos encontrado en [3], especificados en Maude-NPA, quedarían de la siguiente manera:

```
--- Sort Information
 subsort Name Nonce  Key Text < Msg .
 subsort Masterkey  Sessionkey < Key .
 subsort SName UName < Name .
 subsort Name < Public .

--- Nonce operator
 op n : Name Fresh -> Nonce [frozen] .

 --- User names
 ops a b i : -> UName .

 --- Server name
 op s : ->  SName .

--- MKey
 op mkey : Name Name -> Masterkey [frozen comm] .

--- Seskey
 op seskey : Name Name Nonce -> Sessionkey [frozen] .

---encrypt
 op e  : Key Msg -> Msg [frozen] .
 op d : Key Msg -> Msg  [frozen] .

--- Concatenation
 op _;_ : Msg Msg -> Msg [frozen gather (e E)] .
```

Las propiedades algebraicas serían las siguientes:

```
fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
 protecting PROTOCOL-EXAMPLE-SYMBOLS .

 ------------------------------------------------------------
 --- Overwrite this module with the algebraic properties
 --- of your protocol
 ------------------------------------------------------------

 eq d(K:Key, e(K:Key, Z:Msg)) = Z:Msg [metadata "variant"] .
 eq e(K:Key, d(K:Key, Z:Msg)) = Z:Msg [metadata "variant"] .


endfm
```

En este protocolo existen tres roles: Alice, Bob y Servidor.

La especificación del comportamiento del Servidor es la siguiente:

```
    :: r ::
    --- Server's Strand
   [ nil |    -(A ; NA ; B ; NB),
              +(e(mkey(B,s) , NB ; seskey(A , B , n(S,r)) ; A) ;
               e(mkey(A,s) , NA ; seskey(A , B , n(S,r)) ; B) ) , nil]
```

La especificación del comportamiento de Alice es la siguiente:

```
--- Alice's Strand.
  = :: r ::
    [ nil |     +(A ; n(A,r)),
                -(e(mkey(A,s) , n(A,r) ; SK ; B) ; e(SK , NB ; n(A,r))),
                +(e(SK , n(A,r) ; NB)) , nil ]
```

La especificación del comportamiento de Bob es la siguiente:

```
    :: r , r' ::
   --- Bob's Strand.
    [ nil |    -(A ; NA),
               +(A ; NA ; B ; n(B,r')),
                -(e(mkey(B,s) , n(B,r') ; SK ; A) ; MA),
               +(MA ; e(SK , n(B,r) ; NA)),
                -(e(SK , NA ; n(B,r))) , nil ]
```

En este protocolo vamos a considerar tres configuraciones distintas que corresponden a las siguientes propiedades:

1. Una ejecución normal del protocolo (attack-state(0)).
2. Una ejecución donde el intruso averigua la clave de Alice y Bob generada por el servidor (attack-state(1)).
3. Una ejecución donde Bob ha completado el protocolo creyendo que habla con Alice pero no es así (attack-state(2)).

5.7.1. Attack-state(0). Ejecución normal.

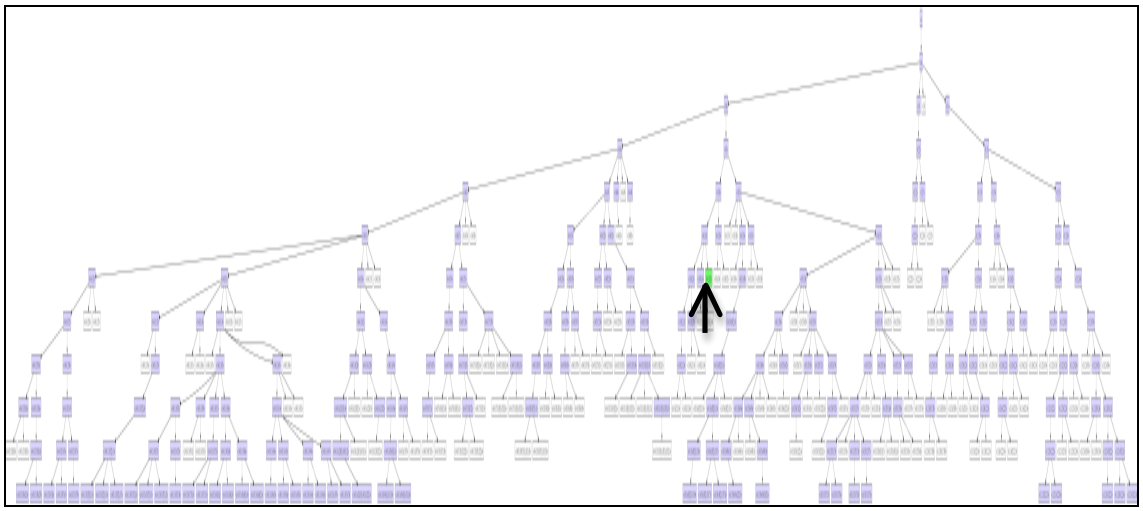
Especificaríamos el attack-state(0) en Maude-NPA de la siguiente manera:

```
eq ATTACK-STATE(0) =
:: r , r' ::
--- Un estado con una ejecución normal del protocolo. añadimos el strand con el último -(M)
   [ nil ,    -(a ; NA),
              +(a ; NA ; b ; n(b,r')),
               -(e(mkey(b,s) , n(b,r') ; SK ; a) ; e(mkey(A,s) , n(A,r) ; SK ; B)),
              +( e(mkey(A,s) , n(A,r) ; SK ; B); e(SK , n(b,r) ; NA)),
               -(e(SK , NA ; n(b,r))) | nil ]
    | | empty
    | | nil
    | | nil
    | | nil
[nonexec] .
```

Como podemos ver, la ejecución del attack-state(0) es una ejecución normal del protocolo. Añadimos el strand con el último mensaje (strand de

Bob). Codificamos la clave de sesión "Kab" como "SK" y el nonce de Alice como "NA".

Mostramos parte del árbol de búsqueda a continuación.



Donde podemos ver que existe una solución, y por lo tanto una ejecución normal del protocolo.

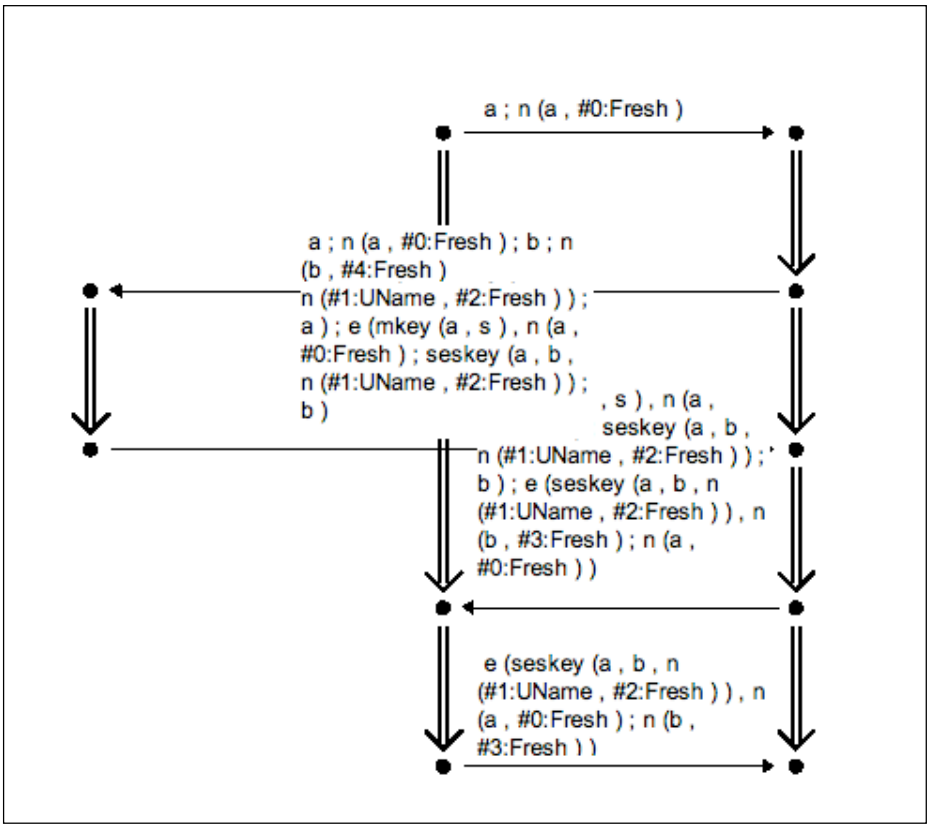
La secuencia de mensajes generados para la solución sería la siguiente:

```
+ ( a ; n ( a , #0:Fresh ) ) ,
 - ( a ; n ( a , #0:Fresh ) ) ,
+ ( a ; n ( a , #0:Fresh ) ; b ; n ( b , #4:Fresh ) ) ,
 - ( a ; n ( a , #0:Fresh ) ; b ; n ( b , #4:Fresh ) ) ,
+ ( e ( mkey ( b , s ) , n ( b , #4:Fresh ) ;
    seskey ( a , b , n ( #1:UName , #2:Fresh ) ) ; a ) ;
    e ( mkey ( a , s ) , n ( a , #0:Fresh ) ;
    seskey ( a , b , n ( #1:UName , #2:Fresh ) ) ; b ) ) ,
- ( e ( mkey ( b , s ) , n ( b , #4:Fresh ) ;
    seskey ( a , b , n ( #1:UName , #2:Fresh ) ) ; a ) ;
    e ( mkey ( a , s ) , n ( a , #0:Fresh ) ;
    seskey ( a , b , n ( #1:UName , #2:Fresh ) ) ; b ) ) ,
+ ( e ( mkey ( a , s ) , n ( a , #0:Fresh ) ;
    seskey ( a , b , n ( #1:UName , #2:Fresh ) ) ; b ) ;
    e ( seskey ( a , b , n ( #1:UName , #2:Fresh ) ) , n ( b , #3:Fresh ) ;
    n ( a , #0:Fresh ) ) ) ,
- ( e ( mkey ( a , s ) , n ( a , #0:Fresh ) ;
    seskey ( a , b , n ( #1:UName , #2:Fresh ) ) ; b ) ;
    e ( seskey ( a , b , n ( #1:UName , #2:Fresh ) ) , n ( b , #3:Fresh ) ;
    n ( a , #0:Fresh ) ) ) ,
+ ( e ( seskey ( a , b , n ( #1:UName , #2:Fresh ) ) , n ( a , #0:Fresh ) ;
     n ( b , #3:Fresh ) ) ) ,
- ( e ( seskey ( a , b , n ( #1:UName , #2:Fresh ) ) , n ( a , #0:Fresh ) ;
    n ( b , #3:Fresh ) ) )
```

Que se expresaría en notación informal de la siguiente manera:

```
(1)  A -> B : A, Na
(2)  B -> S  : A, Na, B, Nb'
(3)  S -> B  : E(Kbs:Nb',Kab,A), E(Kas:Na,Kab,B)
(4)  B -> A : E(Kas:Na,Kab,B), E(Kab:Nb,Na)
(5)  A -> B : E(Kab:Na,Nb)
```

La visualización gráfica de la solución obtenida sería la siguiente:



Podemos ver que se produce una ejecución normal del protocolo.

El strand de Alice generado por la herramienta gráfica sería el siguiente:

```
:: #0:Fresh ::
---Alice's Strand
    [ nil | + ( a ; n ( a , #0:Fresh ) ) ,
            - ( e ( mkey ( a , s ) , n ( a , #0:Fresh ) ;
              seskey ( a , b , n ( #1:UName , #2:Fresh ) ) ; b ) ;
              e ( seskey ( a , b , n ( #1:UName , #2:Fresh ) ) , n ( b , #3:Fresh );
              n ( a , #0:Fresh ) ) ) ,
           + ( e ( seskey ( a , b , n ( #1:UName , #2:Fresh ) ) , n ( a , #0:Fresh );
              n ( b , #3:Fresh ) ) ) , nil ] &
```

El strand de Bob generado por la herramienta gráfica sería el siguiente:

```
:: #3:Fresh , #4:Fresh ::
--- Bob's Strand
[ nil | - ( a ; n ( a , #0:Fresh ) ) ,
       + ( a ; n ( a , #0:Fresh ) ; b ; n ( b , #4:Fresh ) ) ,
        - ( e ( mkey ( b , s ) , n ( b , #4:Fresh ) ;
          seskey ( a , b , n ( #1:UName , #2:Fresh ) ) ; a ) ;
          e ( mkey ( a , s ) , n ( a , #0:Fresh ) ;
          seskey ( a , b , n ( #1:UName , #2:Fresh ) ) ; b ) ) ,
        + ( e ( mkey ( a , s ) , n ( a , #0:Fresh ) ;
          seskey ( a , b , n ( #1:UName , #2:Fresh ) ) ; b ) ;
          e ( seskey ( a , b , n ( #1:UName , #2:Fresh ) ), n ( b , #3:Fresh );
          n ( a , #0:Fresh ) ) ) ,
       - ( e ( seskey ( a , b , n ( #1:UName , #2:Fresh ) ) , n ( a , #0:Fresh );
          n ( b , #3:Fresh ) ) ) , nil ]
```

El strand del Servidor generado por la herramienta gráfica sería el siguiente:

```
:: #2:Fresh ::
---Server's Strand
[ nil | - ( a ; n ( a , #0:Fresh ) ; b ; n ( b , #4:Fresh ) ) ,
        + ( e ( mkey ( b , s ) , n ( b , #4:Fresh ) ;
            seskey ( a , b , n ( #1:UName , #2:Fresh ) ) ; a ) ;
            e ( mkey ( a , s ) , n ( a , #0:Fresh ) ;
            seskey ( a , b , n ( #1:UName , #2:Fresh ) ) ; b ) ) , nil ] &
```

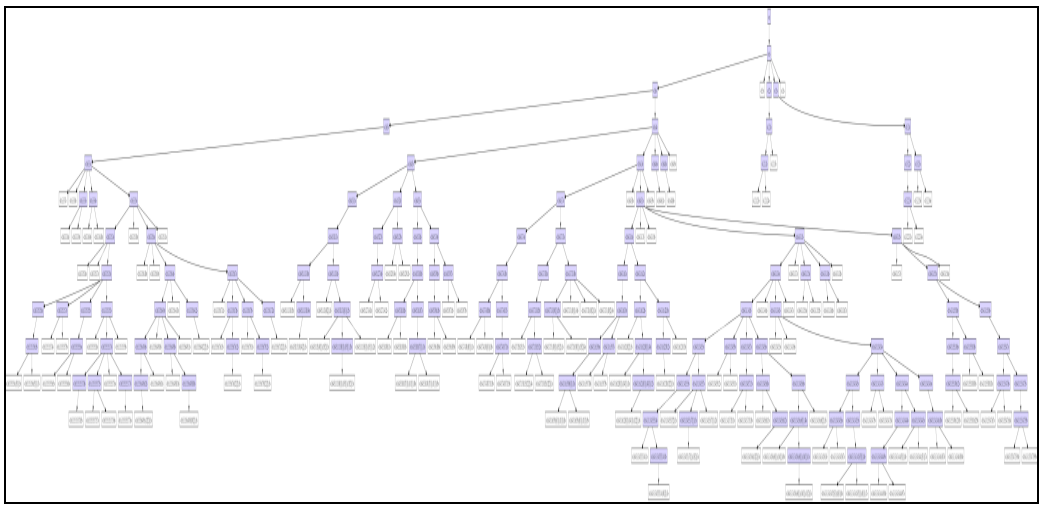
5.7.2. Attack-state(1). Comprobación de secreto.

La implementación del attack-state(1) en Maude-NPA sería:

```
eq ATTACK-STATE(1) =
:: r,r' ::
--- Un estado con un secreto por descubrir
   [ nil ,  -(a ; NA),
            +(a ; NA ; b ; n(b,r')),
             -(e(mkey(b,s) , n(b,r') ; SK ; a) ; MA),
            +(MA ; e(SK , n(b,r) ; NA)),
             -(e(SK , NA ; n(b,r)))  |  nil ]
    | | SK inI
    | | nil
    | | nil
    | | nil
[nonexec] .
```

En este caso indicamos que el intruso podría conocer la clave de sesión entre Alice y Bob generada por el Servidor.

El árbol de búsqueda que se obtiene al ejecutar la herramienta gráfica sería el siguiente:



Podemos observar que no existen soluciones. Por lo tanto es seguro ante un ataque de obtención de la clave de sesión.

5.7.3. Attack-state(2). Comprobación de autenticación.

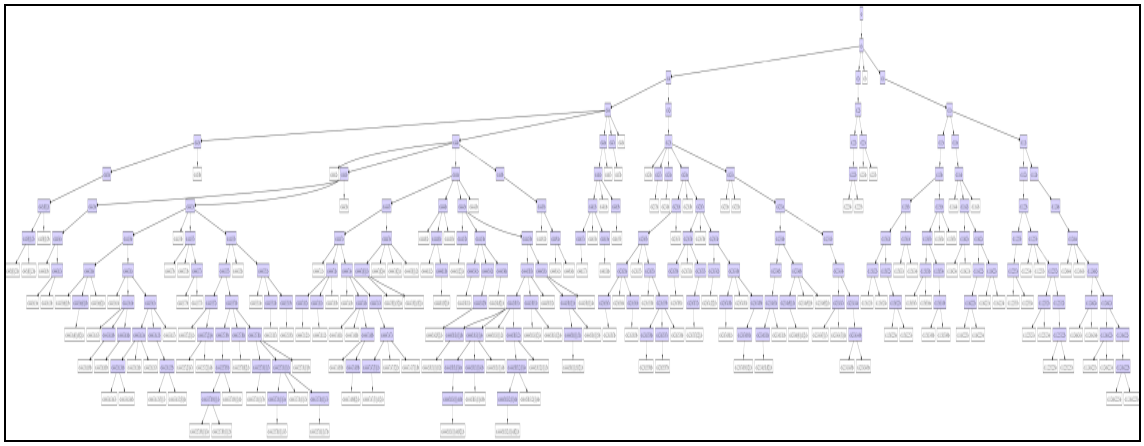
La implementación del attack-state(2) usando Maude-NPA sería:

```
eq ATTACK-STATE(2) =
--- Un estado con una ejecución normal pero con patrón de autenticación.
:: r,r' ::
   [ nil ,  -(a ; NA),
            +(a ; NA ; b ; n(b,r')),
             -(e(mkey(b,s) , n(b,r') ; SK ; a) ; e(mkey(A,s) , NA ; SK ; B)),
            +( e(mkey(A,s) , NA ; SK ; B); e(SK , n(b,r) ; NA)),
             -(e(SK , NA ; n(b,r))) | nil ]
|| empty
|| nil
|| nil
|| never
*** Pattern for authentication
   (:: R:FreshSet ::
   [ nil |    +(a ; NA),
                          -(e(mkey(a,s) , NA ; SK ; b) ; e(SK , n(b,r) ; NA)),
                          +(e(SK , NA ; n(b,r))), nil ]
    & S:StrandSet || K:IntruderKnowledge)

[nonexec] .
```

En este caso usaríamos el strand de Alice como patrón de autenticación.

El árbol de búsqueda que se obtiene al ejecutar la herramienta gráfica sería el siguiente.



Podemos observar que no existen soluciones. Por lo tanto el protocolo es seguro ante un ataque de autenticación.

5.8. Woo and Lam Authentication Protocol.

Woo y Lam proponen un protocolo que proporciona una vía de autenticación del iniciador del protocolo, Alice, a quien ofrece una respuesta, Bob.

El protocolo utiliza criptografía de claves simétricas y un servidor de confianza, con el que Alice y Bob puedan compartir las claves simétricas. La descripción informal del protocolo proporcionada en la sección 6.3.10 de [1], pág. 50 es la siguiente:

(1) A → B : A
(2) B → A : Nb
(3) A → B : E(Kas:Nb)
(4) B → S : E(Kbs:A,E(Kas:Nb))
(5) S → B : E(Kbs:Nb)

Alice inicia el protocolo mediante el envío de su identidad a Bob. Éste responde enviando un nonce "NB" recién generado. Alice encripta el nonce de Bob con la clave "Kas" y la devuelve a Bob. Posteriormente Bob concatena la respuesta de Alice con la identidad de Alice, encriptada con la clave "Kbs" y la envía al Servidor. El Servidor envía "NB" de nuevo a Bob encriptado bajo "Kbs". Entonces, Bob compara el nonce que recibe del Servidor con el que le envió a Alice. Si coinciden, entonces Bob se garantiza que el iniciador del protocolo es, de hecho, quien afirmó ser en la primera etapa del protocolo.

Los subtipos y operadores del protocolo, especificados en Maude-NPA quedarían de la siguiente manera:

```
--- Sort Information
 subsort Name Nonce  Key Text < Msg .
 subsort Masterkey  Sessionkey < Key .
 subsort SName UName < Name .
 subsort Name < Public .

--- Nonce operator
 op n : Name Fresh -> Nonce [frozen] .

 --- User names
 ops a b i z g nm : -> UName .

 --- Server name
 op s : ->  SName .

--- MKey
 op mkey : Name Name -> Masterkey [frozen comm] .

--- Seskey
 op seskey : Name Name Nonce -> Sessionkey [frozen] .

---encrypt
 op e  : Key Msg -> Msg [frozen] .
 op d : Key Msg -> Msg  [frozen] .

--- Concatenation
 op _;_ : Msg Msg -> Msg [frozen gather (e E)] .
```

Las propiedades algebraicas serían las siguientes:

```
fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
 protecting PROTOCOL-EXAMPLE-SYMBOLS .

 ------------------------------------------------------------
 --- Overwrite this module with the algebraic properties
 --- of your protocol
```

```
---------------------------------------------------------

eq d(K:Key, e(K:Key, Z:Msg)) = Z:Msg [metadata "variant"] .
eq e(K:Key, d(K:Key, Z:Msg)) = Z:Msg [metadata "variant"] .


endfm
```

En este protocolo existen tres roles: Servidor, Alice y Bob.

La especificación del comportamiento del Servidor es la siguiente:

```
    :: r ::
    --- Server's Strand
   [ nil |  -(e(mkey(B,s) , A ; e(mkey(A,s) , NB))),
            +(e(mkey(B,s) , NB)) , nil]
```

La especificación del comportamiento de Alice es la siguiente:

```
--- Alice's Strand.
  = :: nil ::
    [ nil |  +(A),
             -(NB),
             +(e(mkey(A,s) , NB)) , nil ]
```

La especificación del comportamiento de Bob es la siguiente:

```
    :: r  ::
--- Bob's Strand.
    [ nil |  -(A),
           +(n(B,r)),
            -(MA),
            +(e(mkey(B,s) , A ; MA)),
            -(e(mkey(B,s) , n(B,r))) , nil ]
```

En este protocolo vamos a considerar una configuración que corresponde a la siguiente propiedad:

1. Una ejecución normal del protocolo (attack-state(0)).

Solo hemos considerado esta configuración (ejecución normal del protocolo) porque durante su ejecución y análisis del resultado hemos observado que no es seguro ante un ataque de autenticación, con lo cual no consideramos necesario ejecutar una configuración para esa propiedad ya que ha quedado constatada en la ejecución normal.

5.8.1. Attack-state(0). Ejecución normal.

La especificación del attack-state(0) en Maude-NPA sería la siguiente:

```
eq ATTACK-STATE(0) =
:: r ::
--- Un estado con una ejecución normal del protocolo. añadimos el strand con el ultimo -(M)
   [ nil ,  -(a),
            +(n(b,r)),
            -(MA),
           +(e(mkey(b,s) , a ; MA)),
            -(e(mkey(b,s) , n(b,r))) | nil ]
    | | empty
```

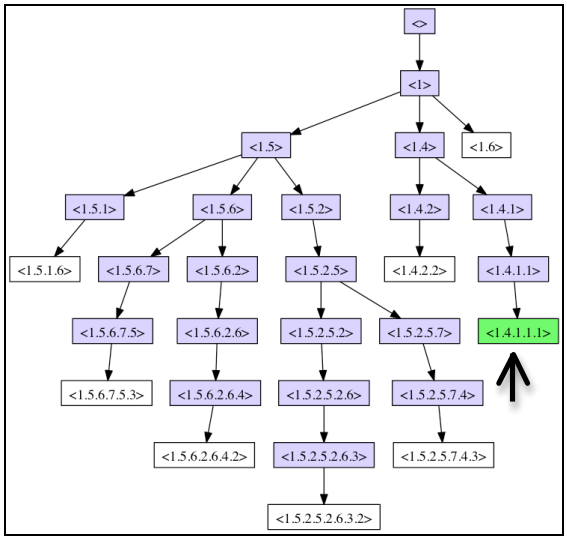
```
        | | nil
        | | nil
        | | nil
      [nonexec] .
```

Incluimos sólo el strand de Bob, sin requerir información sobre que conoce el intruso y la herramienta rellenará el resto.

Procederíamos a ejecutar el attack-state(0) mediante la herramienta gráfica de análisis.

Como podemos ver, la ejecución del attack-state(0) es una ejecución normal del protocolo.

El árbol de búsqueda que se obtiene sería el siguiente.



Donde podemos ver que existen una solución.

La secuencia de mensajes generados para la solución sería la siguiente:

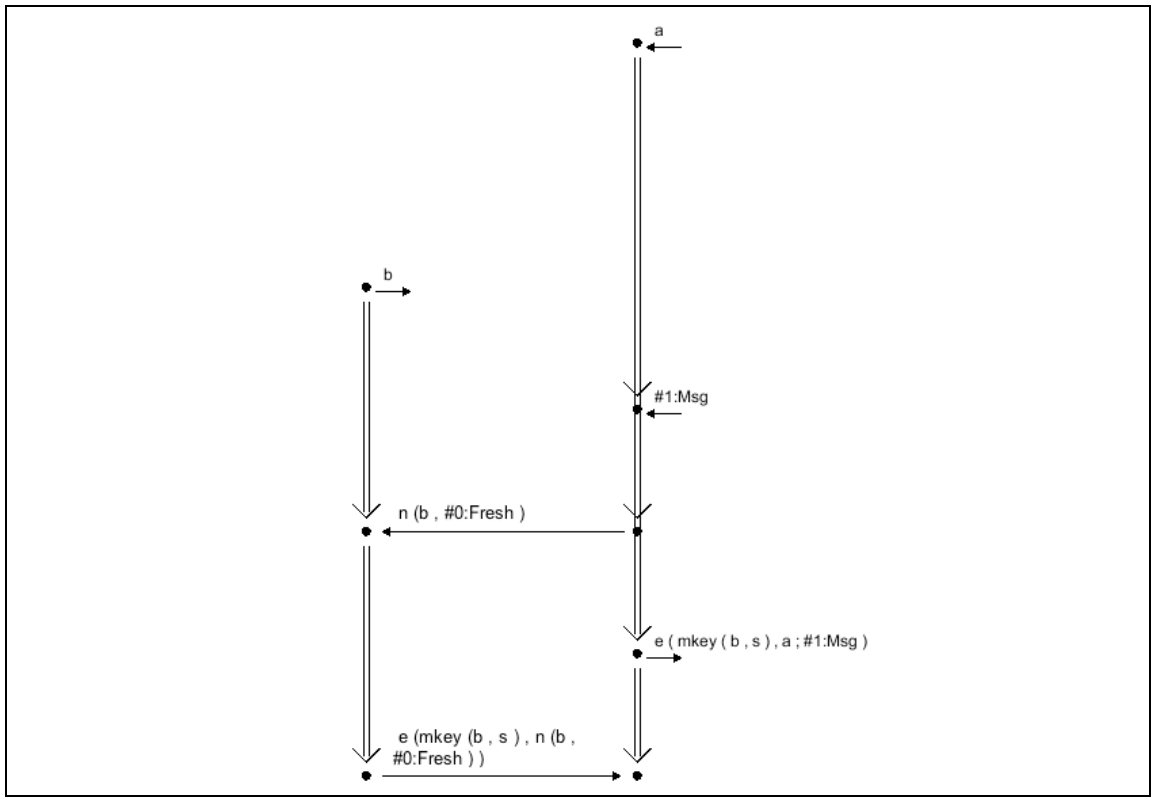
```
        - ( a ) ,
        + ( n ( b , #0:Fresh ) ) ,
        generatedByIntruder ( #1:Msg ) ,
        - ( #1:Msg ) ,
        + ( e ( mkey ( b , s ) , a ; #1:Msg ) ) ,
        + ( b ) ,
        - ( n ( b , #0:Fresh ) ) ,
        + ( e ( mkey ( b , s ) , n ( b , #0:Fresh ) ) ) ,
        - ( e ( mkey ( b , s ) , n ( b , #0:Fresh ) ) )
```

La secuencia de mensajes en notación informal, quedaría de la siguiente manera:

(1) Z(A) → B : A
(2) S → Z(A) : B
(3) Z(A) → B : E(Kas:Nb)
(4) B → S : Nb
(5) B → Z(S) : E(Kbs:A,E(Kas:Nb))
(6) S → B : E(Kbs:Nb)

Podemos observar que este protocolo no es seguro ante un ataque de autenticación, ya que Bob se piensa que está hablando con Alice, cuando realmente no lo está haciendo, por lo tanto no es necesario ejecutar un ataque de comprobación de autenticación.

La visualización gráfica de la obtenida sería la siguiente:



El strand de Alice generado por la herramienta gráfica sería el siguiente:

```
:: nil ::
[ nil | + ( b ) ,
        - ( n ( b , #0:Fresh ) ) ,
        + ( e ( mkey ( b , s ) , n ( b , #0:Fresh ) ) ) , nil ] &
```

El strand de Bob generado por la herramienta gráfica sería el siguiente:

```
:: #0:Fresh ::
--- Bob's Strand
[ nil | - ( a ) ,
        + ( n ( b , #0:Fresh ) ) ,
        - ( #1:Msg ) ,
        + ( e ( mkey ( b , s ) , a ; #1:Msg ) ) ,
        - ( e ( mkey ( b , s ) , n ( b , #0:Fresh ) ) ) , nil ]
```

6. Protocolos de autenticación repetida de clave simétrica.

Algunos protocolos permiten que las claves puedan ser reutilizadas en más de una sesión. Estos son protocolos en dos partes. La primera parte consiste en que un participante , Alice, obtiene un ticket para la comunicación con un segundo participante, Bob. El ticket contiene generalmente una clave de sesión y se cifra de modo que sólo el receptor, Bob, puede descifrarlo. En la segunda parte del protocolo, Alice presenta el ticket a Bob cuando se quiere comunicar, y puede hacer esto en varias ocasiones (hasta que el ticket expira). Éstos protocolos generalmente se llaman protocolos de autenticación repetida.

6.1. Kao Chow Repeated Authentication Protocol.

Kao y Chow proponen un protocolo de autenticación repetida que no es susceptible a los ataques del protocolo de Neuman-Stubblebine.

En este protocolo Kas y Kbs son claves cuyos valores son conocidos inicialmente solo por Alice y el Servidor, y Bob y el Servidor respectivamente.

La descripción informal del protocolo proporcionada en la sección 6.5.4 de [1], pág. 58, es la siguiente:

(1) A→S : A,B,Na
(2) S→B : E(Kas:A,B,Na,Kab),E(Kbs:A,B,Na,Kab)
(3) B→A : E(Kas:A,B,Na,Kab),E(Kab:Na),Nb
(4) A→B : E(Kab:Nb)

Na y Nb son nonces para la autenticación mútua y para verificar la autenticidad de la clave Kab.

Los mensajes 3 y 4 son una repetición de la autenticación: después de que los mensajes 1 y 2 se hayan completado con éxito, 3 y 4 pueden reproducirse repetidamente por Bob antes de iniciar una comunicación secreta con Alice, encriptada con la clave de sesión Kab. Para el tema de la autenticación repetida también se puede consultar el protocolo de Neumann-Stubblebine.

Este protocolo ha sido diseñado para prevenir los ataques "freshness" en la parte de autenticación repetida del protocolo de Neumann-Stubblebine. Realmente, el nonce Na, en el cifrado del mensaje 2, evita que una clave compartida, por ser reutilizada, se vea comprometida después de otra ejecución del protocolo.

Este protocolo debe garantizar el secreto de "Kab": en cada sesión, el valor de "Kab", debe saberse únicamente por los participantes, Alice, Bob y el Servidor.

Cuando Alice recibe la clave "Kab" en el mensaje 3, esta clave debe haber sido emitida en la misma sesión por el Servidor con el que Alice ha comenzado a comunicarse en el mensaje 1.

El protocolo debe asegurar la mutua autenticación de Alice y Bob.

Los subtipos y operadores del protocolo, especificados en Maude-NPA, quedarían de la siguiente manera:

```
--- Sort Information
 subsort Name Nonce Enc Key < Msg .
 subsort Masterkey  Sessionkey < Key .
 subsort SName UName < Name .
 subsort Name < Public .

--- Nonce operator
 op n : Name Fresh -> Nonce [frozen] .

 --- User names
 ops a b i : -> UName .

 --- Server name
 op s : ->  SName .

 --- MKey
 op mkey : Name Name -> Masterkey [frozen] .

 --- Seskey
 op seskey : Name Name Nonce -> Sessionkey [frozen] .

 ---encrypt
 op e : Key Msg -> Msg [frozen] .
 op d : Key Msg -> Msg  [frozen] .

--- successor

 op p : Msg -> Msg [frozen] .


--- Concatenation
 op _;_ : Msg Msg -> Msg [frozen gather (e E)] .
```

Las propiedades algebraicas serían las siguientes:

```
fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
 protecting PROTOCOL-EXAMPLE-SYMBOLS .

 ------------------------------------------------------------
 --- Overwrite this module with the algebraic properties
 --- of your protocol
 ------------------------------------------------------------

 eq d(K:Key, e(K:Key, Z:Msg)) = Z:Msg [metadata "variant"] .
 eq e(K:Key, d(K:Key, Z:Msg)) = Z:Msg [metadata "variant"] .


endfm
```

Este protocolo sufre cuando la clave de sesión se ve comprometida (como en el protocolo de Needham Schroeder con el ataque de Denning-Sacco).

En este protocolo existen tres roles: Servidor, Alice y Bob.

La especificación del comportamiento del Servidor es la siguiente:

```
:: r ::
--- Server's Strand
[ nil | -(A ; B ; NA),
        +(e(mkey(A,s), A ; B ; NA ;
          seskey(A , B , n(s,r))) ;
          e(mkey(B,s) , A ; B ; NA ; seskey(A , B , n(s,r)))), nil ]
```

La especificación del comportamiento de Alice es la siguiente:

```
--- Alice's Strand
= :: r ::
 [ nil | +(A ; B ; n(A,r)),
         -(e(mkey(A,s), A ; B ; n(A,r) ; SK) ; e(SK, n(A,r)) ; NB),
         +(e(SK, NB)) , nil ]
```

La especificación del comportamiento de Bob es la siguiente:

```
--- Bob's Strand
:: r ::
[ nil | -(MA ; e(mkey(B,s), A ; B ; NA ; SK)) ,
        +(MA ; e(SK, NA) ; n(B,r)),
         -(e(SK, n(B,r))), nil ]
```

En este protocolo vamos a considerar dos configuraciones distintas que corresponden a la siguientes propiedades:

1. Una ejecución normal del protocolo (attack-state(0)).
2. Una ejecución donde el intruso averigua la clave de Alice y Bob generada por el servidor (attack-state(1)).
3. Una ejecución donde Bob ha completado el protocolo creyendo que habla con Alice pero no es así (attack-state(2)).

6.1.1. Attack-state(0). Ejecución normal.

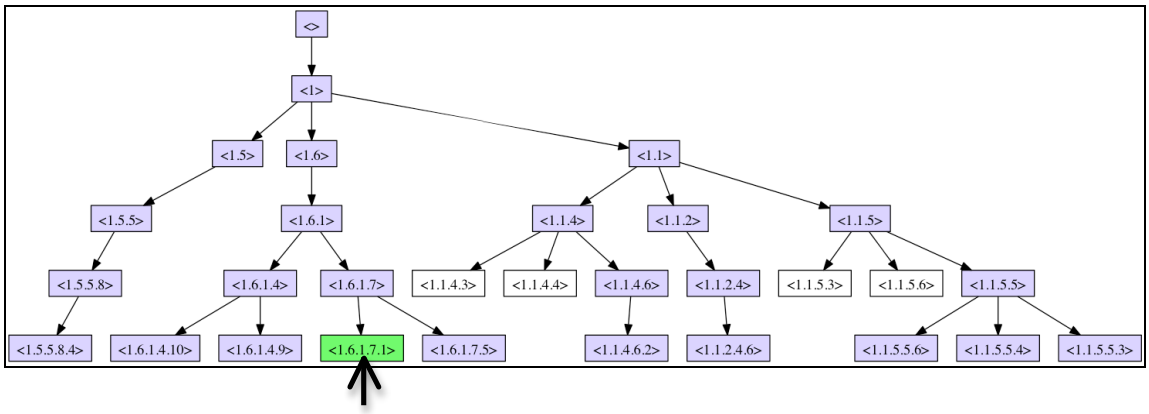
Especificaríamos el attack-state(0) en Maude-NPA de la siguiente manera:

```
eq ATTACK-STATE(0) =
   :: r ::
   [ nil , -(MA ; e(mkey(b,s), a ; b ; NA ; SK)) ,
           +(MA ; e(SK, NA) ; n(b,r)),
           -(e(SK, n(b,r))) | nil ]
   | | empty
   | | nil
   | | nil
   | | nil
[nonexec] .
```

En este patrón de ataque se incluye sólo el participante del protocolo que responde. En el patrón se indica que el participante ha terminado su ejecución y el nonce recibido del iniciador del protocolo es desconocido y se

representa con una variable “NA”. La clave de sesión “Kab” se representa como “SK”.

Mostramos parte del árbol de búsqueda a continuación:



Donde podemos ver que se encuentra una solución.

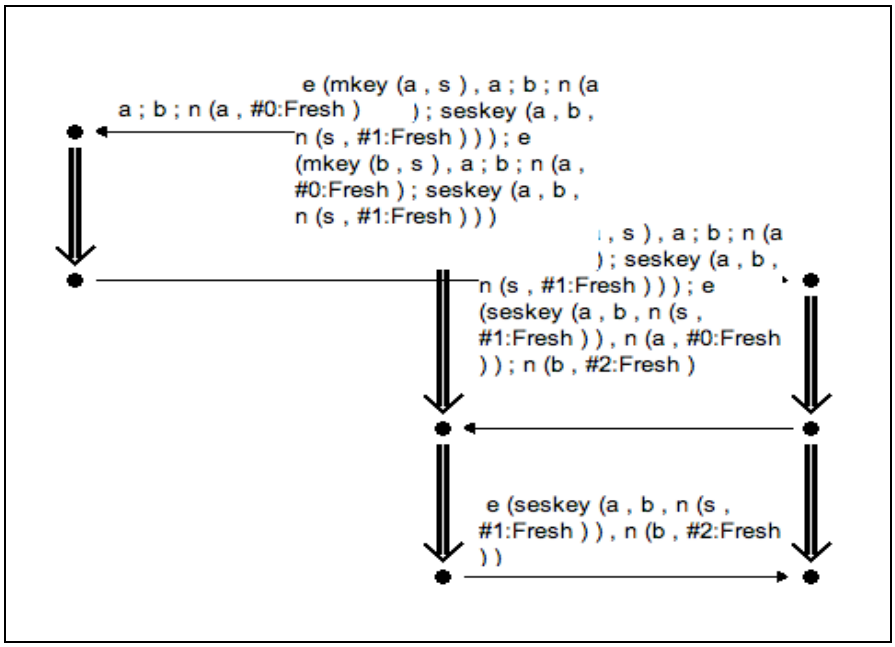
La secuencia de mensajes en la ejecución sería la siguiente:

```
+ ( a ; b ; n ( a , #0:Fresh ) ) ,
 - ( a ; b ; n ( a , #0:Fresh ) ) ,
+ ( e ( mkey ( a , s ) , a ; b ; n ( a , #0:Fresh ) ;
    seskey ( a , b , n ( s , #1:Fresh ) ) ) ;
    e ( mkey ( b , s ) , a ; b ; n ( a , #0:Fresh ) ;
    seskey ( a , b , n ( s , #1:Fresh ) ) ) ) ,
- ( e ( mkey ( a , s ) , a ; b ; n ( a , #0:Fresh ) ;
   seskey ( a , b , n ( s , #1:Fresh ) ) ) ;
   e ( mkey ( b , s ) , a ; b ; n ( a , #0:Fresh ) ;
   seskey ( a , b , n ( s , #1:Fresh ) ) ) ) ,
+ ( e ( mkey ( a , s ) , a ; b ; n ( a , #0:Fresh ) ;
    seskey ( a , b , n ( s , #1:Fresh ) ) ) ;
    e ( seskey ( a , b , n ( s , #1:Fresh ) ) , n ( a , #0:Fresh ) ) ;
    n ( b , #2:Fresh ) ) ,
- ( e ( mkey ( a , s ) , a ; b ; n ( a , #0:Fresh ) ;
    seskey ( a , b , n ( s , #1:Fresh ) ) ) ;
    e ( seskey ( a , b , n ( s , #1:Fresh ) ) , n ( a , #0:Fresh ) ) ;
    n ( b , #2:Fresh ) ) ,
+ ( e ( seskey ( a , b , n ( s , #1:Fresh ) ) , n ( b , #2:Fresh ) ) ) ,
 - ( e ( seskey ( a , b , n ( s , #1:Fresh ) ) , n ( b , #2:Fresh ) ) )
```

La interpretación de esta secuencia de ataque en notación informal sería la siguiente:

(1) A→S : A,B,Na
(2) S→B : E(Kas:A,B,Na,Kab),E(Kbs:A,B,Na,Kab)
(3) B→A : E(Kas:A,B,Na,Kab),E(Kab:Na),Nb
(4) A→B : E(Kab:Nb)

La visualización de los strands de la ejecución de la solución en el attack-state(0) sería la siguiente



Donde podemos ver que se produce una ejecución normal.

El strand del Servidor generado por la herramienta gráfica sería el siguiente:

```
:: #1:Fresh ::
[ nil | - ( a ; b ; n ( a , #0:Fresh ) ) ,
      + ( e ( mkey ( a , s ) , a ; b ; n ( a , #0:Fresh ) ;
          seskey ( a , b , n ( s , #1:Fresh ) ) ) ;
          e ( mkey ( b , s ) , a ; b ; n ( a , #0:Fresh ) ;
          seskey ( a , b , n ( s , #1:Fresh ) ) ) ) , nil ] &
```

El strand de Alice generado por la herramienta gráfica sería el siguiente:

```
 :: #0:Fresh ::
[ nil | + ( a ; b ; n ( a , #0:Fresh ) ) ,
        - ( e ( mkey ( a , s ) , a ; b ; n ( a , #0:Fresh ) ;
            seskey ( a , b , n ( s , #1:Fresh ) ) ) ;
            e ( seskey ( a , b , n ( s , #1:Fresh ) ) , n ( a , #0:Fresh ) ) ;
            n ( b , #2:Fresh ) ) ,
        + ( e ( seskey ( a , b , n ( s , #1:Fresh ) ) , n ( b , #2:Fresh ) ) ) , nil ] &
```

El strand de Bob generado por la herramienta gráfica sería el siguiente:

```
:: #2:Fresh ::
[ nil | - ( e ( mkey ( a , s ) , a ; b ; n ( a , #0:Fresh ) ;
          seskey ( a , b , n ( s , #1:Fresh ) ) ) ;
          e ( mkey ( b , s ) , a ; b ; n ( a , #0:Fresh ) ;
          seskey ( a , b , n ( s , #1:Fresh ) ) ) ) ,
      + ( e ( mkey ( a , s ) , a ; b ; n ( a , #0:Fresh ) ;
          seskey ( a , b , n ( s , #1:Fresh ) ) ) ;
          e ( seskey ( a , b , n ( s , #1:Fresh ) ) , n ( a , #0:Fresh ) ) ;
          n ( b , #2:Fresh ) ) ,
      - ( e ( seskey ( a , b , n ( s , #1:Fresh ) ) , n ( b , #2:Fresh ) ) ) , nil ]
```

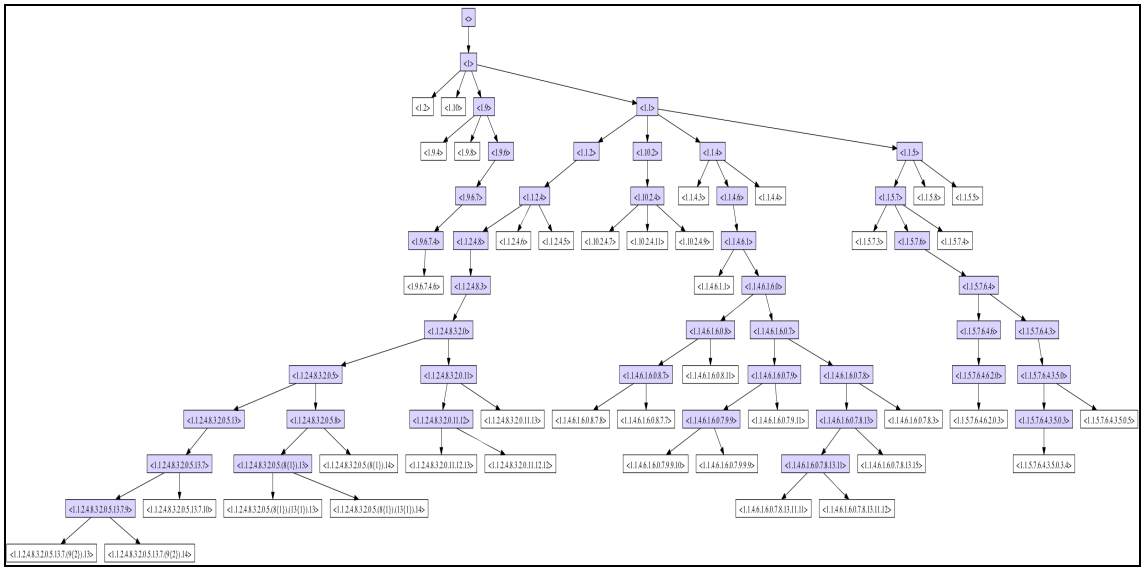
6.1.2. Attack-state(1). Comprobación de un secreto.

El attack-state(1) especificado en Maude-NPA sería el siguiente:

```
eq ATTACK-STATE(1) =
:: r ::
[ nil, -(MA ; e(mkey(b,s), a ; b ; NA ; SK)) ,
      +(MA ; e(SK, NA) ; n(b,r)),
      -(e(SK, n(b,r)))
        | nil ]
| | SK inI
| | nil
| | nil
| | nil
[nonexec] .
```

En este caso indicamos que el intruso podría conocer la clave de sesión entre Alice y Bob generada por el Servidor. Hemos definido como "MA" el mensaje "e(mkey(a,s), a ; b ; NA ; SK)"

El árbol de búsqueda que se obtiene usando la herramienta gráfica sería el siguiente:



Donde podemos ver que no existe una solución. Por lo tanto, es seguro ante un ataque de obtención de la clave de sesión. Con esto demostramos y confirmamos que este protocolo ha sido diseñado para prevenir los ataques "freshness" en la parte de autenticación repetida del protocolo de Neumann-Stubblebine, evitando que una clave compartida, por ser reutilizada, se vea comprometida después de otra ejecución del protocolo.

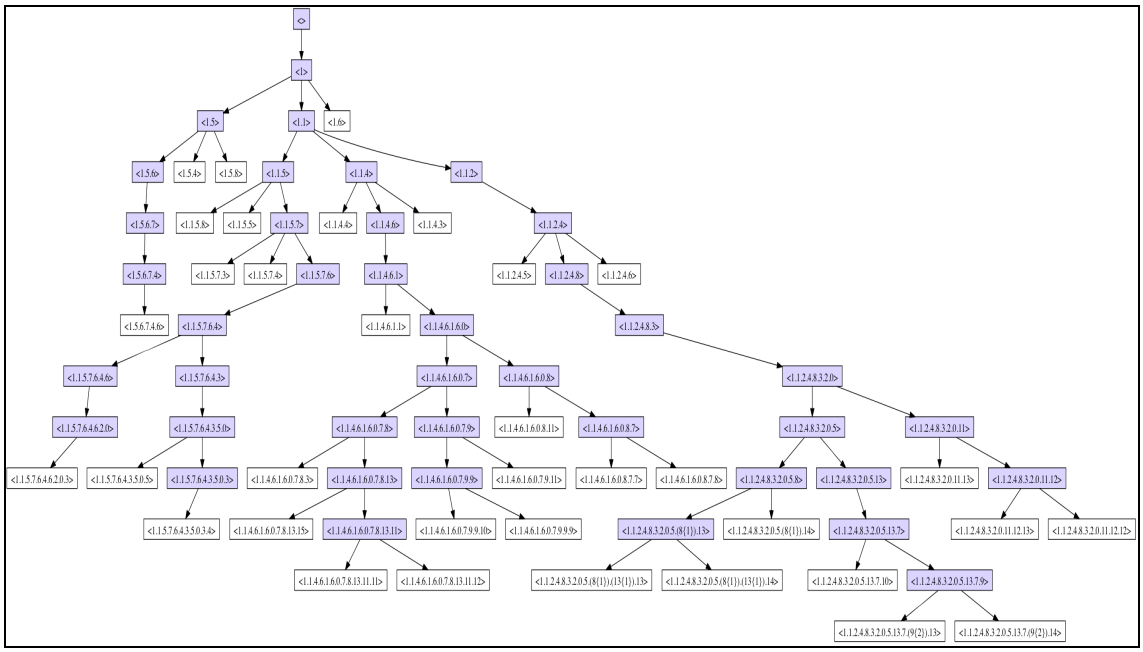
6.1.3. Attack-state(2). Comprobación de autenticación.

Especificaríamos el attack-state(2) en Maude-NPA de la siguiente manera:

```
eq ATTACK-STATE(2) =
:: r ::
---Bob's Strand
   [ nil , -(MA ; e(mkey(b,s), a ; b ; NA ; SK)) ,
         +(MA ; e(SK, NA) ; n(b,r)),
         -(e(SK, n(b,r))) | nil ]
    | | empty
    | | nil
    | | nil
    | | never
   *** Pattern for authentication
   (:: R:FreshSet ::
   [ nil | +(A ; B ; NA),
           -(MA ; e(SK, NA) ; n(b,r)),
           +(e(SK, n(b,r))), nil ]
    & S:StrandSet | | K:IntruderKnowledge)
  [nonexec] .
```

En este caso usaríamos el strand de Alice como patrón de autenticación. Hemos definido como "MA" el mensaje "e(mkey(a,s), a ; b ; NA ; SK)"

El árbol de búsqueda que se obtiene sería el siguiente:



Donde podemos ver que no existe una solución. Por lo tanto el protocolo es seguro ante un ataque de autenticación.

6.2. Kao Chow Repeated Authentication Protocol with Handshake Key.

Kao y Chow proponen usar una clave únicamente para el intercambio ("handshake key" o clave de intercambio). Como la clave simplemente es para el intercambio, la he definido como "Kt:Key". "Kas" y "Kbs" son claves simétricas cuyos valores son inicialmente conocidas solamente por Alice y el Servidor, y Bob y el Servidor respectivamente.

"Na" y "Nb" son nonces para la autenticación mutua y para verificar la autenticidad de la clave del sistema (Kab).

La descripción informal del protocolo proporcionada en la sección 6.5.4 de [1], pág. 59 es la siguiente:

(1) A→S : A,B,Na
(2) S→B : E(Kas:A,B,Na,Kab,Kt),E(Kbs:A,B,Na,Kab,Kt)
(3) B→A : E(Kas:A,B,Na,Kab),E(Kt:Na,Kab),Nb
(4) A→B : E(Kt:Na,Kab)

Los mensajes 3 y 4 son la autenticación repetida: después de que estos mensajes 1 y 2 se hayan completado con éxito, y 3 y 4 pueden reproducirse sucesivamente por Bob antes de comenzar una comunicación con Alice, encriptada con la clave de sesión "Kab". Este protocolo ha sido diseñado para prevenir el ataque en la parte de autenticación repetida del protocolo de Neumann-Stubblebine. De hecho, el nonce "Na" en el cifrado del mensaje 2 previene el compromiso de la clave compartida después de otra ejecución del protocolo y que pueda ser reutilizada.

La especificación en Maude-NPA sería la siguiente:

```
--- Sort Information
 subsort Name Nonce Enc Key < Msg .
 subsort Masterkey  Sessionkey < Key .
 subsort SName UName < Name .
 subsort Name < Public .

--- Nonce operator
 op n : Name Fresh -> Nonce [frozen] .
 op t : Name Fresh -> Nonce [frozen] . ---Nonce del server

 --- User names
 ops a b i : -> UName .

 --- Server name
 op s : ->  SName .

 --- MKey
 op mkey : Name Name -> Masterkey [frozen] .

 --- Seskey
 op seskey : Name Name Nonce -> Sessionkey [frozen] .

--- Encoding operators for public/private encryption
 op pk : Key Msg -> Enc [frozen] .
 op sk : Key Msg -> Enc [frozen] .

 ---encrypt
 op e : Key Msg -> Msg [frozen] .
 op d : Key Msg -> Msg  [frozen] .

--- Concatenation
 op _;_ : Msg Msg -> Msg [frozen gather (e E)] .
```

Las propiedades algebraicas serían las siguientes:

```
fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
  protecting PROTOCOL-EXAMPLE-SYMBOLS .

  ------------------------------------------------------------
  --- Overwrite this module with the algebraic properties
  --- of your protocol
  ------------------------------------------------------------

 eq d(K:Key, e(K:Key, Z:Msg)) = Z:Msg [metadata "variant"] .
 eq e(K:Key, d(K:Key, Z:Msg)) = Z:Msg [metadata "variant"] .


endfm
```

En este protocolo existen tres roles: Servidor, Alice y Bob.

La especificación del comportamiento del Servidor es la siguiente:

```
  :: r,r' ::
   --- Server's Strand
  [ nil | -(A ; B ; NA),
          +(e(mkey(A,s), A ; B ; NA ;
            seskey(A , B , n(s,r)) ; Kt) ;
            e(mkey(B,s) , A ; B ; NA ; seskey(A , B , n(s,r)) ; Kt)), nil ]
```

La especificación del comportamiento de Alice es la siguiente:

```
  --- Alice's Strand
  :: r ::
  [ nil | +(A ; B ; n(A,r)),
           -(e(mkey(A,s), A ; B ; n(A,r) ; SK) ; e(Kt, n(A,r) ; SK) ; NB),
           +(e(Kt, n(A,r) ; SK)) , nil ]
```

La especificación del comportamiento de Bob es la siguiente:

```
  --- Bob's Strand
  :: r ::
  [ nil | -(MA ; e(mkey(B,s), A ; B ; NA ; SK ; Kt)) ,
          +(MA ; e(Kt, NA ; SK) ; n(B,r)),
           -(e(Kt, NA ; SK)) , nil ]
```

En este protocolo vamos a considerar una configuración que corresponde a la siguiente propiedad:

1. Una ejecución normal del protocolo (attack-state(0)).

Sólo hemos considerado una configuración ya que durante la ejecución del attack-state(0) hemos podido observar que la clave del sistema ("Kab"), se ve comprometida también, por eso no hemos ejecutado una configuración específica para esa propiedad ya que quedaba plasmada en la configuración elegida.

En este patrón de ataque (attack-state(0)) se incluye sólo el participante del protocolo que responde. En el patrón se indica que el participante ha terminado su ejecución (la barra de progreso está al final del strand) y el nonce recibido del iniciador del protocolo es desconocido y se representa

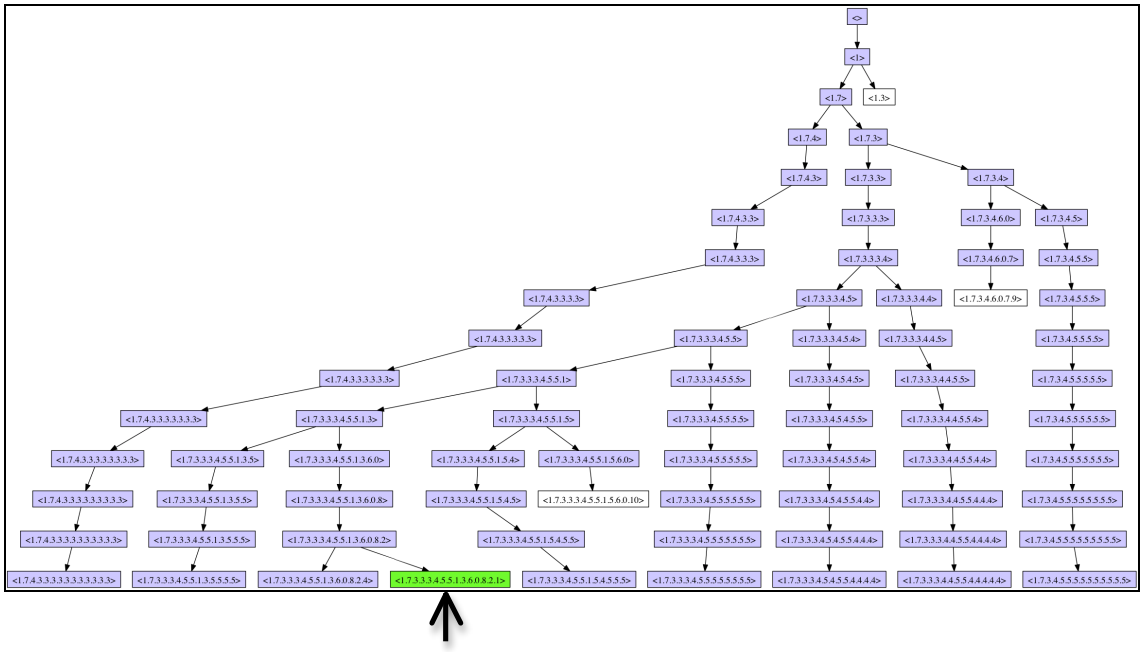
con una variable “NA”. La clave de sesión “Kab” se representa como “SK”. Usamos una clave únicamente para el intercambio. Como la clave simplemente es para el intercambio la he definido como “Kt:Key”.

6.2.1. Attack-state(0). Ejecución normal.

Especificaríamos el attack-state(0) en Maude-NPA de la siguiente manera:

```
eq ATTACK-STATE(0) =
   :: nil ::
   [ nil , -(MA ; e(mkey(b,s), a ; b ; NA ; SK ; Kt)) ,
         +(MA ; e(Kt, n(a,r) ; SK) ; n(b,r)),
         -(e(Kt, NA ; SK)) | nil ]
   || empty
   || nil
   || nil
   || nil
[nonexec] .
```

Mostramos parte del árbol de búsqueda a continuación:



Donde podemos ver que existe una solución, y por lo tanto una ejecución normal del protocolo.

La secuencia de mensajes en la ejecución sería la siguiente:

```
generatedByIntruder ( #3:UName ; i ; #4:Nonce ) ,
+ ( mkey ( i , s ) ) ,
- ( #3:UName ; i ; #4:Nonce ) ,
+ ( e ( mkey ( #3:UName , s ) , #3:UName ; i ; #4:Nonce ;
    seskey ( #3:UName , i , n ( s , #5:Fresh ) ) ;
    mkey ( b , s ) ) ; e ( mkey ( i , s ) , #3:UName ; i ; #4:Nonce ;
    seskey ( #3:UName , i , n ( s , #5:Fresh ) ) ; mkey ( b , s ) ) ) ,
- ( e ( mkey ( #3:UName , s ) , #3:UName ; i ; #4:Nonce ;
    seskey ( #3:UName , i , n ( s , #5:Fresh ) ) ;
```

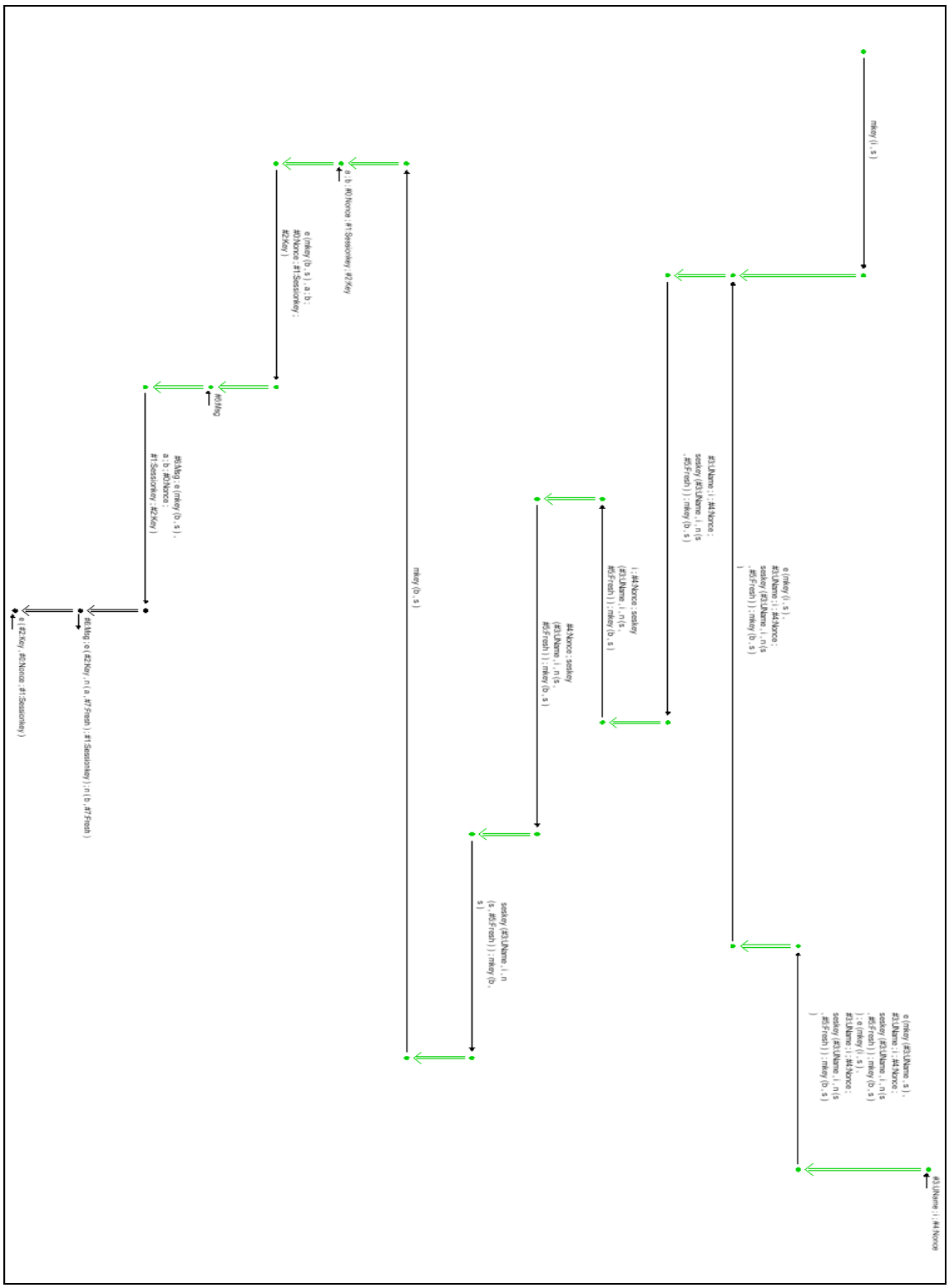
```
        mkey ( b , s ) ) ; e ( mkey ( i , s ) , #3:UName ; i ; #4:Nonce ;
        seskey ( #3:UName , i , n ( s , #5:Fresh ) ) ; mkey ( b , s ) ) ) ,
    + ( e ( mkey ( i , s ) , #3:UName ; i ; #4:Nonce ;
        seskey ( #3:UName , i , n ( s , #5:Fresh ) ) ; mkey ( b , s ) ) ) ,
    resuscitated ( mkey ( i , s ) ) ,
    - ( mkey ( i , s ) ) ,
    - ( e ( mkey ( i , s ) , #3:UName ; i ; #4:Nonce ;
        seskey ( #3:UName , i , n ( s , #5:Fresh ) ) ; mkey ( b , s ) ) ) ,
    + ( #3:UName ; i ; #4:Nonce ;
        seskey ( #3:UName , i , n ( s , #5:Fresh ) ) ; mkey ( b , s ) ) ,
    - ( #3:UName ; i ; #4:Nonce ;
        seskey ( #3:UName , i , n ( s , #5:Fresh ) ) ; mkey ( b , s ) ) ,
    + ( i ; #4:Nonce ; seskey ( #3:UName , i , n ( s , #5:Fresh ) ) ; mkey ( b , s ) ) ,
     - ( i ; #4:Nonce ; seskey ( #3:UName , i , n ( s , #5:Fresh ) ) ; mkey ( b , s ) ) ,
    + ( #4:Nonce ; seskey ( #3:UName , i , n ( s , #5:Fresh ) ) ; mkey ( b , s ) ) ,
     - ( #4:Nonce ; seskey ( #3:UName , i , n ( s , #5:Fresh ) ) ; mkey ( b , s ) ) ,
    + ( seskey ( #3:UName , i , n ( s , #5:Fresh ) ) ; mkey ( b , s ) ) ,
     - ( seskey ( #3:UName , i , n ( s , #5:Fresh ) ) ; mkey ( b , s ) ) ,
    + ( mkey ( b , s ) ) ,
    generatedByIntruder ( e ( #2:Key , #0:Nonce ; #1:Sessionkey ) ) ,
    generatedByIntruder ( a ; b ; #0:Nonce ; #1:Sessionkey ; #2:Key ) ,
     - ( mkey ( b , s ) ) ,
     - ( a ; b ; #0:Nonce ; #1:Sessionkey ; #2:Key ) ,
    + ( e ( mkey ( b , s ) , a ; b ; #0:Nonce ; #1:Sessionkey ; #2:Key ) ) ,
     - ( e ( mkey ( b , s ) , a ; b ; #0:Nonce ; #1:Sessionkey ; #2:Key ) ) ,
     - ( e ( mkey ( b , s ) , a ; b ; #0:Nonce ; #1:Sessionkey ; #2:Key ) ) ,
    + ( e ( mkey ( b , s ) , a ; b ; #0:Nonce ; #1:Sessionkey ; #2:Key ) ;
        e ( mkey ( b , s ) , a ; b ; #0:Nonce ; #1:Sessionkey ; #2:Key ) ) ,
     - ( e ( mkey ( b , s ) , a ; b ; #0:Nonce ; #1:Sessionkey ; #2:Key ) ;
        e ( mkey ( b , s ) , a ; b ; #0:Nonce ; #1:Sessionkey ; #2:Key ) ) ,
    + ( e ( mkey ( a , s ) , a ; b ; #0:Nonce ; #1:Sessionkey ) ;
        e ( #2:Key , n ( a , #6:Fresh ) ; #1:Sessionkey ) ; n ( b , #6:Fresh ) ) ,
    - ( e ( #2:Key , #0:Nonce ; #1:Sessionkey ) )
```

La interpretación de esta secuencia de ataque en notación informal sería la siguiente:

```
(1) I → A       : Kis
(2) A → I       : A,I,Na
(3) S→B         : E(Kas:A,B,Na,Kab,Kt),E(Kbs:A,B,Kab,Kt)
(4) I→A         : E(Kis:A,I,Na,Kai,Kbs)
(5) I→A         : A,I,Na,Kai,Kbs
(6) I→A         : I,Nb,Kai,Kbs
(7) I→A         : Nb,Kai,Kbs
(8) I→A         : Kai,Kbs
(9) I→A         : Kbs
(10) I(A)→B     : E(Kt,Na,Kab)
(11) I(S)→B     : Kbs → Generado por el intruso
(12) A→S        : A,B,Na,Kab,Kt
(13) I(S) → B   : E(Kbs:A,B,Na,Kab,Kt)
(14) I(S) → B   : E(Kbs:A,B,Na,Kab,Kt),E(Kbs:A,B,Na,Kab,Kt)
(15) B→A        : E(Kas:A,B,Na,Kab),E(Kt:Na,Kab,Nb)
(16) A→B        : E(Kt:Na,Kab)
```

En el análisis de este ataque podemos ver también que la clave del sistema (Kab), se ve comprometida también, por ejemplo en "seskey (#3:UName , i , n (s , #5:Fresh))".

La visualización de los strands de la ejecución de la solución en el attack-state(0) sería la siguiente:



Los strands de los participantes generados por la herramienta gráfica serían los siguientes:

```
:: nil :: [ nil | + ( mkey ( i , s ) ) , nil ] &

:: nil :: [ nil | - ( mkey ( b , s ) ) , - ( a ; b ; #0:Nonce ; #1:Sessionkey ; #2:Key ) , + ( e ( mkey ( b , s ) , a ; b ; #0:Nonce ; #1:Sessionkey ; #2:Key ) ) , nil ] &

:: nil :: [ nil | - ( mkey ( i , s ) ) , - ( e ( mkey ( i , s ) , #3:UName ; i ; #4:Nonce ; seskey ( #3:UName , i , n ( s , #5:Fresh ) ) ; mkey ( b , s ) ) ) , + ( #3:UName ; i ; #4:Nonce ; seskey ( #3:UName , i , n ( s , #5:Fresh ) ) ; mkey ( b , s ) ) , nil ] &

:: nil :: [ nil | - ( e ( mkey ( b , s ) , a ; b ; #0:Nonce ; #1:Sessionkey ; #2:Key ) ) , - ( #6:Msg ) , + ( #6:Msg ; e ( mkey ( b , s ) , a ; b ; #0:Nonce ; #1:Sessionkey ; #2:Key ) ) , nil ] &

:: nil :: [ nil | - ( i ; #4:Nonce ; seskey ( #3:UName , i , n ( s , #5:Fresh ) ) ; mkey ( b , s ) ) , + ( #4:Nonce ; seskey ( #3:UName , i , n ( s , #5:Fresh ) ) ; mkey ( b , s ) ) , nil ] &

:: nil :: [ nil | - ( #6:Msg ; e ( mkey ( b , s ) , a ; b ; #0:Nonce ; #1:Sessionkey ; #2:Key ) ) , + ( #6:Msg ; e ( #2:Key , n ( a , #7:Fresh ) ; #1:Sessionkey ) ; n ( b , #7:Fresh ) ) , - ( e ( #2:Key , #0:Nonce ; #1:Sessionkey ) ) , nil ] &

:: nil :: [ nil | - ( #3:UName ; i ; #4:Nonce ; seskey ( #3:UName , i , n ( s , #5:Fresh ) ) ; mkey ( b , s ) ) , + ( i ; #4:Nonce ; seskey ( #3:UName , i , n ( s , #5:Fresh ) ) ; mkey ( b , s ) ) , nil ] &

:: nil :: [ nil | - ( #4:Nonce ; seskey ( #3:UName , i , n ( s , #5:Fresh ) ) ; mkey ( b , s ) ) , + ( seskey ( #3:UName , i , n ( s , #5:Fresh ) ) ; mkey ( b , s ) ) , nil ] &

:: nil :: [ nil | - ( e ( mkey ( #3:UName , s ) , #3:UName ; i ; #4:Nonce ; seskey ( #3:UName , i , n ( s , #5:Fresh ) ) ; mkey ( b , s ) ) ; e ( mkey ( i , s ) , #3:UName ; i ; #4:Nonce ; seskey ( #3:UName , i , n ( s , #5:Fresh ) ) ; mkey ( b , s ) ) ) , + ( e ( mkey ( i , s ) , #3:UName ; i ; #4:Nonce ; seskey ( #3:UName , i , n ( s , #5:Fresh ) ) ; mkey ( b , s ) ) ) , nil ] &

:: nil :: [ nil | - ( seskey ( #3:UName , i , n ( s , #5:Fresh ) ) ; mkey ( b , s ) ) , + ( mkey ( b , s ) ) , nil ] &

:: #5:Fresh , #8:Fresh :: [ nil | - ( #3:UName ; i ; #4:Nonce ) , + ( e ( mkey ( #3:UName , s ) , #3:UName ; i ; #4:Nonce ; seskey ( #3:UName , i , n ( s , #5:Fresh ) ) ; mkey ( b , s ) ) ; e ( mkey ( i , s ) , #3:UName ; i ; #4:Nonce ; seskey ( #3:UName , i , n ( s , #5:Fresh ) ) ; mkey ( b , s ) ) ) , nil ]
```

6.3. Kao Chow Repeated Authentication Protocols with tickets.

El protocolo de Kao y Chow se puede extender para su uso con tickets. En este caso he definido "Ta" como "mkey(A,t)", donde "t : → SName" . Esto lo he hecho por la interpretación que hago de la definición que encuentro en el protocolo Kerberos versión 5 : "G es denominado como un Servidor del cual se pueden obtener tickets y provee claves para la comunicación entre clientes C y servidores S". Lo que hemos hecho ha sido adaptarlo a una clave entre un cliente, Alice, y un Servidor "t", que sería el Servidor del cual se pueden obtener tickets. La descripción informal del protocolo proporcionada en la sección 6.5.4 de [1], pág. 59, es la siguiente:

(1) A→S : A,B,Na
(2) S→B : E(Kas:A,B,Na,Kab,Kt),E(Kbs:A,B,Na,Kab,Kt)
(3) B→A : E(Kas:A,B,Na,Kab),E(Kt:Na,Kab),Nb,E(Kbs:A,B,Ta,Kab)
(4) A→B : E(Kt:Na,Kab),E(Kbs:A,B,Ta,Kab)

La implementación en Maude-NPA del protocolo y de los operadores utilizados sería la siguiente. En este caso, hemos representado al ticket "Ta" como "mkey(A,t)". Definimos también la clave de intercambio Kt como "Kt:Key".

```
--- Sort Information
 subsort Name Nonce Enc Key < Msg .
 subsort Masterkey  Sessionkey < Key .
 subsort SName UName < Name .
 subsort Name < Public .

--- Nonce operator
 op n : Name Fresh -> Nonce [frozen] .
 op t : Name Fresh -> Nonce [frozen] . ---Nonce del server

 --- User names
 ops a b i : -> UName .

 --- Server name
 op s : ->  SName .
 op t : -> SName .

 --- MKey
 op mkey : Name Name -> Masterkey [frozen] .

 --- Seskey
 op seskey : Name Name Nonce -> Sessionkey [frozen] .

--- Encoding operators for public/private encryption
 op pk : Key Msg -> Enc [frozen] .
 op sk : Key Msg -> Enc [frozen] .

 ---encrypt
 op e : Key Msg -> Msg [frozen] .
 op d : Key Msg -> Msg  [frozen] .

--- Concatenation
 op _;_ : Msg Msg -> Msg [frozen gather (e E)] .
```

Las propiedades algebraicas serían las siguientes:

```
fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
 protecting PROTOCOL-EXAMPLE-SYMBOLS .

 ------------------------------------------------------------
 --- Overwrite this module with the algebraic properties
 --- of your protocol
 ------------------------------------------------------------

eq d(K:Key, e(K:Key, Z:Msg)) = Z:Msg [metadata "variant"] .
eq e(K:Key, d(K:Key, Z:Msg)) = Z:Msg [metadata "variant"] .

endfm
```

En este protocolo existen tres roles: Servidor, Alice y Bob.

La especificación del comportamiento del Servidor es la siguiente:

```
   :: r,r' ::
   --- Server's Strand
  [ nil  |  -(A ; B ; NA),
          +(e(mkey(A,s), A ; B ; NA ;
             seskey(A , B , n(s,r)) ; Kt) ;
             e(mkey(B,s) , A ; B ; NA ; seskey(A , B , n(s,r)) ; Kt)), nil ]
```

La especificación del comportamiento de Alice es la siguiente:

```
--- Alice's Strand
= :: r ::
 [ nil | +(A ; B ; n(A,r)),
         -(e(mkey(A,s), A ; B ; n(A,r) ; SK) ; e(Kt, n(A,r) ; SK) ; NB ; MB'),
         +(e(Kt, n(A,r) ; SK) ;  MB) , nil ]
```

La especificación del comportamiento de Bob es la siguiente:

```
--- Bob's Strand
:: r ::
[ nil | -(MA' ; e(mkey(B,s), A ; B ; NA ; SK ; Kt)) ,
        +(MA ; e(Kt, NA ; SK) ; n(B,r) ; e(mkey(B,s), A ; B ; TA ; SK)),
        -(e(Kt, NA ; SK) ; e(mkey(B,s), A ; B ; TA ; SK)) , nil ]
```

En este protocolo vamos a considerar una configuración que corresponde a la siguiente propiedad:

1. Una ejecución normal del protocolo (attack-state(0)).
2. Una ejecución donde el intruso averigua la clave de Alice y Bob generada por el servidor (attack-state(1)).
3. Una ejecución donde Bob ha completado el protocolo creyendo que habla con Alice pero no es así (attack-state(2)).

6.3.1. Attack-state(0). Ejecución normal.

Especificaríamos el attack-state(0) en Maude-NPA de la siguiente manera:

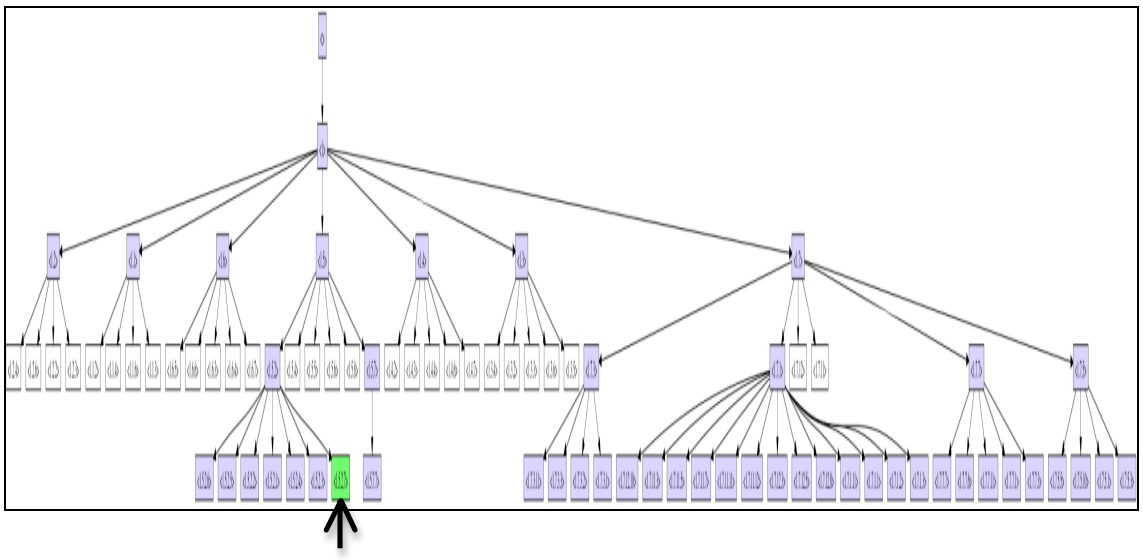
```
eq ATTACK-STATE(0) =
   :: nil ::
   [ nil , -(MA' ; e(mkey(b,s), a ; b ; NA ; SK ; Kt)) ,
           +(MA ; e(Kt, n(a,r) ; SK) ; n(b,r) ; e(mkey(B,s), A ; B ; Kt ; SK)),
           -(e(Kt, NA ; SK) ; e(mkey(b,s), a ; b ; Kt ; SK)) | nil ]
   || empty
   || nil
   || nil
   || nil
[nonexec] .
```

En este patrón de ataque se incluye sólo el participante del protocolo que responde. En el patrón se indica que el participante ha terminado su ejecución y el nonce recibido del iniciador del protocolo es desconocido y se representa con una variable "NA". La clave de sesión "Kab" se representa como "SK". También hemos representado al ticket "Ta" como "mkey(A,t)". Usamos una clave únicamente para el intercambio.

Como la clave simplemente es para el intercambio la he definido como "Kt:Key". Hemos representado las siguientes cadenas con variables de tipo "Msg":

- "e(mkey(B,s), A ; B ; mkey(A,t) ; SK)" como la variable "**MB**".
- "e(mkey(A,s), A ; B ; n(A,r) ; SK)" como la variable "**MA**".
- "e(mkey(A,s), A ; B ; n(A,r) ; SK ; Kt)" como la variable "**MA'**".

Mostramos parte del árbol de búsqueda a continuación:



Donde podemos ver que existe una solución, y por lo tanto una ejecución normal del protocolo.

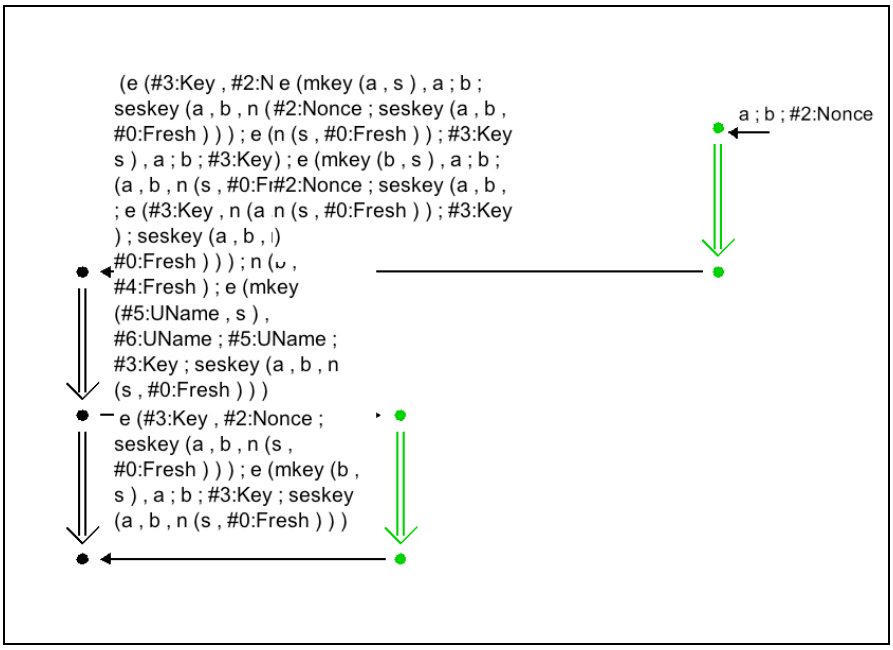
La secuencia de mensajes en la ejecución sería la siguiente:

```
generatedByIntruder ( a ; b ; #2:Nonce ) ,
 - ( a ; b ; #2:Nonce ) ,
+ ( e ( mkey ( a , s ) , a ; b ; #2:Nonce ;
    seskey ( a , b , n ( s , #0:Fresh ) ) ; #3:Key ) ;
    e ( mkey ( b , s ) , a ; b ; #2:Nonce ;
    seskey ( a , b , n ( s , #0:Fresh ) ) ; #3:Key ) ) ,
- ( e ( mkey ( a , s ) , a ; b ; #2:Nonce ;
   seskey ( a , b , n ( s , #0:Fresh ) ) ; #3:Key ) ;
   e ( mkey ( b , s ) , a ; b ; #2:Nonce ;
   seskey ( a , b , n ( s , #0:Fresh ) ) ; #3:Key ) ) ,
+ ( ( e ( #3:Key , #2:Nonce ; seskey ( a , b , n ( s , #0:Fresh ) ) ) ;
     e ( mkey ( b , s ) , a ; b ; #3:Key ; seskey ( a , b , n ( s , #0:Fresh ) ) ) ) ) ;
     e ( #3:Key , n ( a , #4:Fresh ) ; seskey ( a , b , n ( s , #0:Fresh ) ) ) ;
     n ( b , #4:Fresh ) ;
     e ( mkey ( #5:UName , s ) , #6:UName ; #5:UName ; #3:Key ;
     seskey ( a , b , n ( s , #0:Fresh ) ) ) ) ) ,
- ( ( e ( #3:Key , #2:Nonce ; seskey ( a , b , n ( s , #0:Fresh ) ) ) ;
     e ( mkey ( b , s ) , a ; b ; #3:Key ; seskey ( a , b , n ( s , #0:Fresh ) ) ) ) ) ;
     e ( #3:Key , n ( a , #4:Fresh ) ; seskey ( a , b , n ( s , #0:Fresh ) ) ) ;
     n ( b , #4:Fresh ) ;
     e ( mkey ( #5:UName , s ) , #6:UName ; #5:UName ; #3:Key ;
     seskey ( a , b , n ( s , #0:Fresh ) ) ) ) ) ,
+ ( e ( #3:Key , #2:Nonce ; seskey ( a , b , n ( s , #0:Fresh ) ) ) ;
    e ( mkey ( b , s ) , a ; b ; #3:Key ;
    seskey ( a , b , n ( s , #0:Fresh ) ) ) ) ,
- ( e ( #3:Key , #2:Nonce ; seskey ( a , b , n ( s , #0:Fresh ) ) ) ;
    e ( mkey ( b , s ) , a ; b ; #3:Key ;
    seskey ( a , b , n ( s , #0:Fresh ) ) ) )
```

El ataque se expresaría en notación informal de la siguiente manera:

(1) I(A) → S : A,B,Na
(2) S → B : E(Kas:A,B,Na,Kab,Kt),E(Kbs:A,B,Na,Kab,Kt)
(3) B → A : E(Kas:A,B,Na,Kab),E(Kt:Na,Kab),Nb,E(Kbs:A,B,Ta,Kab)
(4) A → B : E(Kt:Na,Kab),E(Kbs:A,B,Ta,Kab)

La visualización de los strands de la ejecución de la solución en el attack-state(0) sería la siguiente:



El strand de Bob generado por la herramienta gráfica sería el siguiente:

```
:: nil ::
[ nil | - ( e ( mkey ( a , s ) , a ; b ; #2:Nonce ;
            seskey ( a , b , n ( s , #0:Fresh ) ) ; #3:Key ) ;
            e ( mkey ( b , s ) , a ; b ; #2:Nonce ;
            seskey ( a , b , n ( s , #0:Fresh ) ) ; #3:Key ) ) ,
        + ( ( e ( #3:Key , #2:Nonce ;
            seskey ( a , b , n ( s , #0:Fresh ) ) ) ;
            e ( mkey ( b , s ) , a ; b ; #3:Key ;
            seskey ( a , b , n ( s , #0:Fresh ) ) ) ) ) ;
            e ( #3:Key , n ( a , #4:Fresh ) ) ;
            seskey ( a , b , n ( s , #0:Fresh ) ) ) ;
            n ( b , #4:Fresh ) ) ;
            e ( mkey ( #5:UName , s ) , #6:UName ; #5:UName ; #3:Key ;
            seskey ( a , b , n ( s , #0:Fresh ) ) ) ) ,
         - ( e ( #3:Key , #2:Nonce ;
            seskey ( a , b , n ( s , #0:Fresh ) ) ) ;
            e ( mkey ( b , s ) , a ; b ; #3:Key ;
            seskey ( a , b , n ( s , #0:Fresh ) ) ) ) , nil ] &
```

El strand de Alice generado por la herramienta gráfica sería el siguiente:

```
:: nil ::
[ nil | - ( ( e ( #3:Key , #2:Nonce ;
                seskey ( a , b , n ( s , #0:Fresh ) ) ) ;
                e ( mkey ( b , s ) , a ; b ; #3:Key ;
                seskey ( a , b , n ( s , #0:Fresh ) ) ) ) ;
                e ( #3:Key , n ( a , #4:Fresh ) ;
                seskey ( a , b , n ( s , #0:Fresh ) ) ) ;
                n ( b , #4:Fresh ) ;
                e ( mkey ( #5:UName , s ) , #6:UName ; #5:UName ; #3:Key ;
                seskey ( a , b , n ( s , #0:Fresh ) ) ) ) ,
              + ( e ( #3:Key , #2:Nonce ;
                  seskey ( a , b , n ( s , #0:Fresh ) ) ) ;
                  e ( mkey ( b , s ) , a ; b ; #3:Key ;
                  seskey ( a , b , n ( s , #0:Fresh ) ) ) ) , nil ] &
```

El strand del Servidor generado por la herramienta gráfica sería el siguiente:

```
:: #1:Fresh , #0:Fresh ::
[ nil | - ( a ; b ; #2:Nonce ) ,
        + ( e ( mkey ( a , s ) , a ; b ; #2:Nonce ;
            seskey ( a , b , n ( s , #0:Fresh ) ) ; #3:Key ) ;
            e ( mkey ( b , s ) , a ; b ; #2:Nonce ;
            seskey ( a , b , n ( s , #0:Fresh ) ) ; #3:Key ) ) , nil ]
```

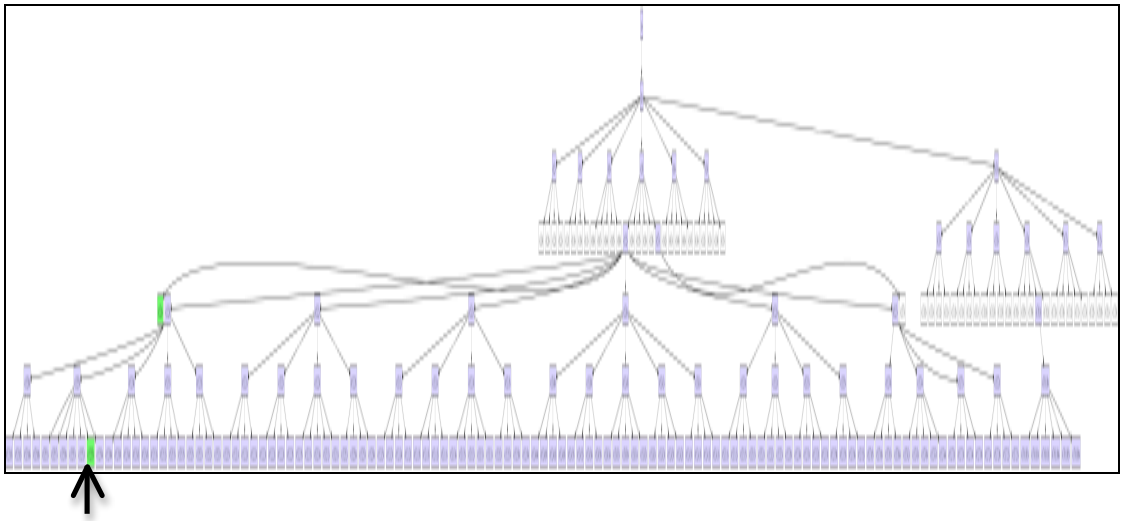
6.3.2. Attack-state(1). Comprobación de secreto.

El attack-state(1) especificado en Maude-NPA sería el siguiente:

```
eq ATTACK-STATE(1) =
:: r ::
 [ nil , -(MA' ; e(mkey(b,s), a ; b ; NA ; SK ; Kt)) ,
        +(MA ; e(Kt, NA ; SK) ; n(b,r) ; e(mkey(b,s), a ; b ; Kt ; SK)),
        -(e(Kt, NA ; SK) ; e(mkey(b,s), a ; b ; Kt ; SK)) | nil ]
| | SK inI
| | nil
| | nil
| | nil
[nonexec] .
```

En este caso indicamos que el intruso podría conocer la clave de sesión entre Alice y Bob generada por el Servidor.

Parte del árbol de búsqueda que se obtiene usando la herramienta gráfica se muestra a continuación.



Donde podemos ver que existe una solución. Por lo tanto no es seguro ante un ataque de obtención de la clave de sesión.

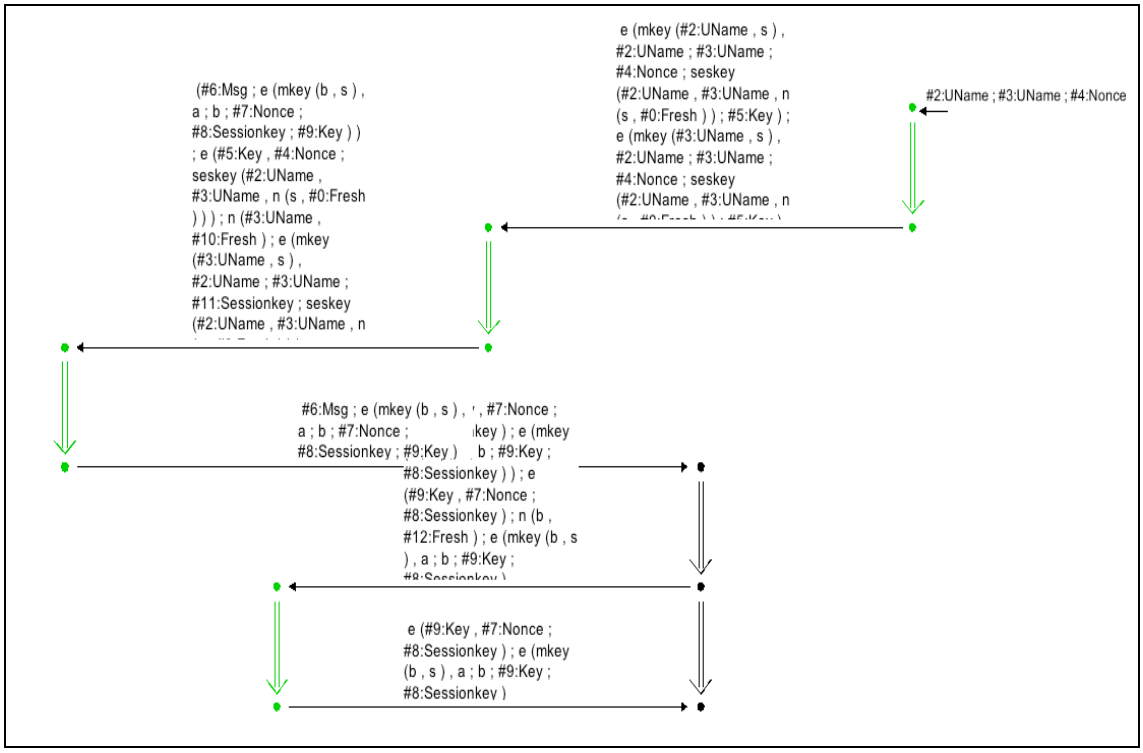
La secuencia de mensajes en la ejecución sería la siguiente

```
generatedByIntruder ( #2:UName ; #3:UName ; #4:Nonce ) ,
- ( #2:UName ; #3:UName ; #4:Nonce ) ,
+ ( e ( mkey ( #2:UName , s ) , #2:UName ; #3:UName ; #4:Nonce ;
    seskey ( #2:UName , #3:UName , n ( s , #0:Fresh ) ) ; #5:Key ) ;
    e ( mkey ( #3:UName , s ) , #2:UName ; #3:UName ; #4:Nonce ;
    seskey ( #2:UName , #3:UName , n ( s , #0:Fresh ) ) ; #5:Key ) ) ,
 generatedByIntruder ( #8:Sessionkey ) ,
- ( e ( mkey ( #2:UName , s ) , #2:UName ; #3:UName ; #4:Nonce ;
    seskey ( #2:UName , #3:UName , n ( s , #0:Fresh ) ) ; #5:Key ) ;
    e ( mkey ( #3:UName , s ) , #2:UName ; #3:UName ; #4:Nonce ;
    seskey ( #2:UName , #3:UName , n ( s , #0:Fresh ) ) ; #5:Key ) ) ,
+ ( ( #6:Msg ; e ( mkey ( b , s ) , a ; b ; #7:Nonce ; #8:Sessionkey ; #9:Key ) ) ;
     e ( #5:Key , #4:Nonce ; seskey ( #2:UName , #3:UName , n ( s , #0:Fresh ) ) ) ;
     n ( #3:UName , #10:Fresh ) ;
     e ( mkey ( #3:UName , s ) , #2:UName ; #3:UName ; #11:Sessionkey ;
     seskey ( #2:UName , #3:UName , n ( s , #0:Fresh ) ) ) ) ,
- ( ( #6:Msg ; e ( mkey ( b , s ) , a ; b ; #7:Nonce ; #8:Sessionkey ; #9:Key ) ) ;
     e ( #5:Key , #4:Nonce ; seskey ( #2:UName , #3:UName , n ( s , #0:Fresh ) ) ) ;
     n ( #3:UName , #10:Fresh ) ;
     e ( mkey ( #3:UName , s ) , #2:UName ; #3:UName ; #11:Sessionkey ;
     seskey ( #2:UName , #3:UName , n ( s , #0:Fresh ) ) ) ) ,
+ ( #6:Msg ; e ( mkey ( b , s ) , a ; b ; #7:Nonce ; #8:Sessionkey ; #9:Key ) ) ,
- ( #6:Msg ; e ( mkey ( b , s ) , a ; b ; #7:Nonce ; #8:Sessionkey ; #9:Key ) ) ,
+ ( ( e ( #9:Key , #7:Nonce ; #8:Sessionkey ) ;
     e ( mkey ( b , s ) , a ; b ; #9:Key ; #8:Sessionkey ) ) ;
     e ( #9:Key , #7:Nonce ; #8:Sessionkey ) ;
     n ( b , #12:Fresh ) ; e ( mkey ( b , s ) , a ; b ; #9:Key ; #8:Sessionkey ) ) ,
- ( ( e ( #9:Key , #7:Nonce ; #8:Sessionkey ) ;
     e ( mkey ( b , s ) , a ; b ; #9:Key ; #8:Sessionkey ) ) ;
     e ( #9:Key , #7:Nonce ; #8:Sessionkey ) ; n ( b , #12:Fresh ) ;
     e ( mkey ( b , s ) , a ; b ; #9:Key ; #8:Sessionkey ) ) ,
+ ( e ( #9:Key , #7:Nonce ; #8:Sessionkey ) ;
     e ( mkey ( b , s ) , a ; b ; #9:Key ; #8:Sessionkey ) ) ,
- ( e ( #9:Key , #7:Nonce ; #8:Sessionkey ) ;
    e ( mkey ( b , s ) , a ; b ; #9:Key ; #8:Sessionkey ) )
```

La representación en notación informal sería la siguiente

(1) I(A)→S : G,Z,Ng
(2) S→I(A) : E(Kgs:G,Z,Ng,Kgz,Kt),E(Kzs:G,Z,Ng,Kgz,Kt)
(3) I(S)→B : E(Kgs:G,Z,Ng,Kgz,Kt), E(Kbs,A,B,Na,Kgz,Kt)
(4) B→I(A) : E(Kt:Ng,Kgz), E(Kbs:A,B,Kt,Kgz), E(Kt:Ng,Kgz) Nb,
E(Kbs:A,B,Kt,Kgz)
(5) (A)→B : E(Kt:Ng,Kgz),E(Kbs:A,B,Kt:Kgz)

La visualización de los strands de la ejecución de la solución en el attack-state(1) sería la siguiente:



6.3.3. Attack-state(2). Comprobación de autenticación.

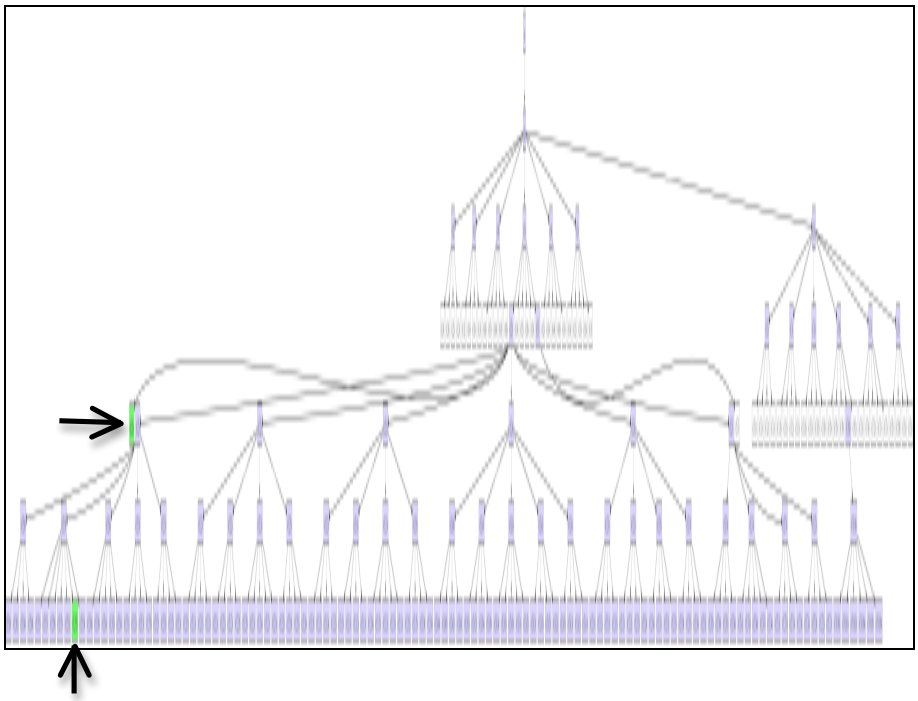
El attack-state(2) especificado en Maude-NPA sería el siguiente:

```
eq ATTACK-STATE(2) =
:: r ::
---Bob's Strand
    [ nil , -(MA' ; e(mkey(b,s), a ; b ; NA ; SK ; Kt)) ,
          +(MA ; e(Kt, NA ; SK) ; n(b,r) ; e(mkey(b,s), a ; b ; Kt ; SK)),
          -(e(Kt, NA ; SK) ; e(mkey(b,s), a ; b ; Kt ; SK)) | nil ]
    | | empty
    | | nil
    | | nil
    | | never
    *** Pattern for authentication
    (:: R:FreshSet ::
    [ nil | +(a ; b ; NA),
           -(MA ; e(Kt, NA ; SK) ; n(b,r) ; e(mkey(B,s), a ; b ; Kt ; SK)),
           +(e(Kt, NA ; SK) ;  e(mkey(b,s), a ; b ; Kt ; SK)), nil ]
```

```
        & S:StrandSet | | K:IntruderKnowledge)
      [nonexec] .
```

En este caso usaríamos el strand de Alice como patrón de autenticación.

Parte del árbol de búsqueda que se obtiene al ejecutar la herramienta gráfica sería el siguiente:



Donde podemos ver que existe dos soluciones. Estas dos soluciones coinciden con las dos soluciones de los ataques anteriores. Podemos decir que no es seguro ante un ataque de autenticación.

La secuencia de mensajes en la ejecución de la **primera solución** sería la siguiente:

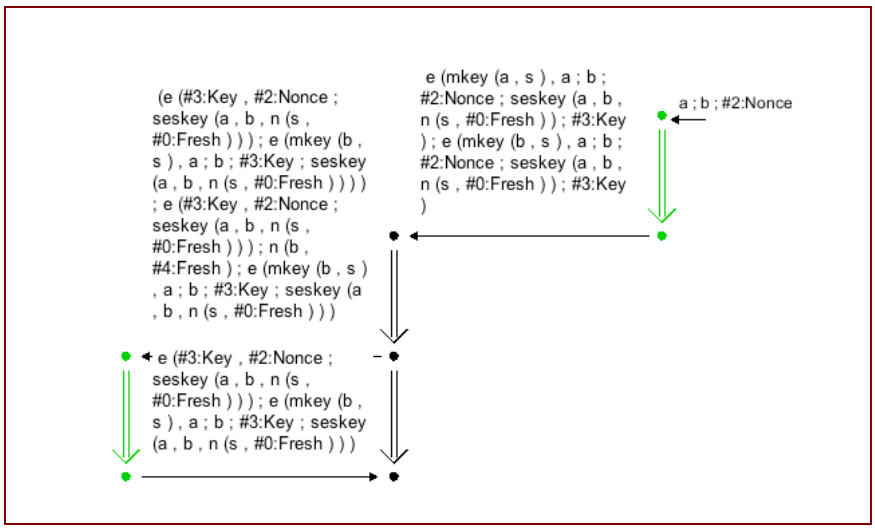
```
generatedByIntruder ( a ; b ; #2:Nonce ) ,
- ( a ; b ; #2:Nonce ) ,
+ ( e ( mkey ( a , s ) , a ; b ; #2:Nonce ;
    seskey ( a , b , n ( s , #0:Fresh ) ) ; #3:Key ) ;
    e ( mkey ( b , s ) , a ; b ; #2:Nonce ; seskey ( a , b , n ( s , #0:Fresh ) ) ; #3:Key ) ) ,
- ( e ( mkey ( a , s ) , a ; b ; #2:Nonce ;
    seskey ( a , b , n ( s , #0:Fresh ) ) ; #3:Key ) ;
    e ( mkey ( b , s ) , a ; b ; #2:Nonce ; seskey ( a , b , n ( s , #0:Fresh ) ) ; #3:Key ) ) ,
+ ( ( e ( #3:Key , #2:Nonce ;
    seskey ( a , b , n ( s , #0:Fresh ) ) ) ;
    e ( mkey ( b , s ) , a ; b ; #3:Key ; seskey ( a , b , n ( s , #0:Fresh ) ) ) ) ;
    e ( #3:Key , #2:Nonce ; seskey ( a , b , n ( s , #0:Fresh ) ) ) ;
    n ( b , #4:Fresh ) ;
    e ( mkey ( b , s ) , a ; b ; #3:Key ; seskey ( a , b , n ( s , #0:Fresh ) ) ) ) ,
- ( ( e ( #3:Key , #2:Nonce ;
    seskey ( a , b , n ( s , #0:Fresh ) ) ) ;
    e ( mkey ( b , s ) , a ; b ; #3:Key ; seskey ( a , b , n ( s , #0:Fresh ) ) ) ) ;
    e ( #3:Key , #2:Nonce ; seskey ( a , b , n ( s , #0:Fresh ) ) ) ;
    n ( b , #4:Fresh ) ;
    e ( mkey ( b , s ) , a ; b ; #3:Key ; seskey ( a , b , n ( s , #0:Fresh ) ) ) ) ,
```

```
+ ( e ( #3:Key , #2:Nonce ; seskey ( a , b , n ( s , #0:Fresh ) ) ) ;
    e ( mkey ( b , s ) , a ; b ; #3:Key ; seskey ( a , b , n ( s , #0:Fresh ) ) ) ) ,
- ( e ( #3:Key , #2:Nonce ; seskey ( a , b , n ( s , #0:Fresh ) ) ) ;
   e ( mkey ( b , s ) , a ; b ; #3:Key ; seskey ( a , b , n ( s , #0:Fresh ) ) ) )
```

La representación en notación informal sería la siguiente:

(1)	I(A)→S : A,B,Na
(2)	S→B : E(Kas:A,B,Na,Kab,Kt),E(Kbs:A,B,Na,Kab,Kt)
(3)	B→A : E(Kas:A,B,Na,Kab), E(Kt:Na,Kab), Nb, E(Kbs:A,B,Ta,Kab)
(4)	A→B : E(Kt:Na,Kab),E(Kbs:A,B,Ta,Kab)

La visualización de los strands de la ejecución sería la siguiente:



La secuencia de mensajes en la ejecución de la **segunda solución** sería la siguiente:

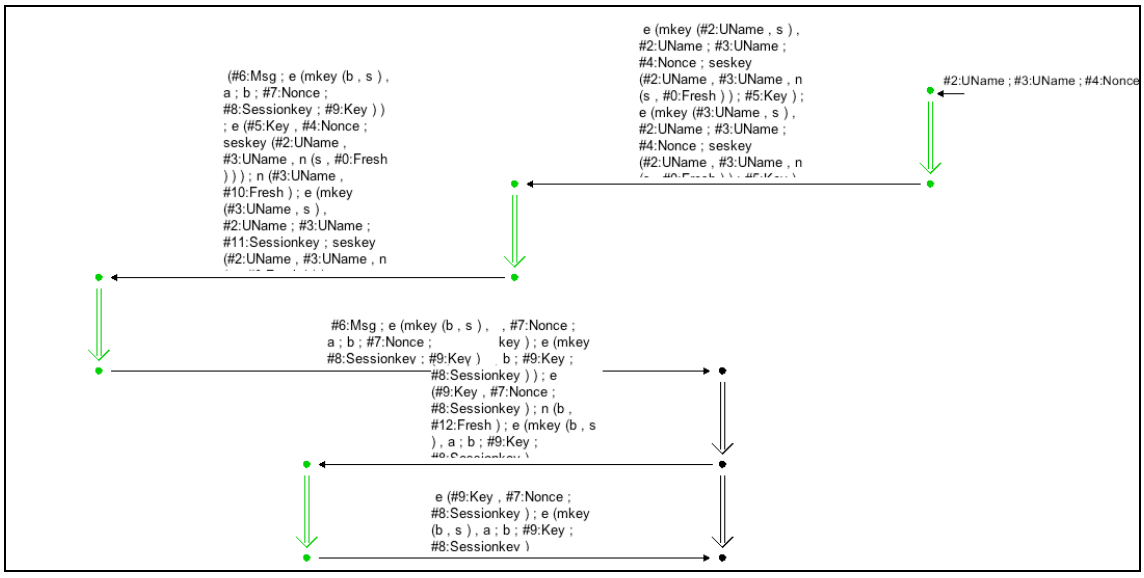
```
generatedByIntruder ( #2:UName ; #3:UName ; #4:Nonce ) ,
- ( #2:UName ; #3:UName ; #4:Nonce ) ,
+ ( e ( mkey ( #2:UName , s ) , #2:UName ; #3:UName ; #4:Nonce ;
   seskey ( #2:UName , #3:UName , n ( s , #0:Fresh ) ) ; #5:Key ) ;
   e ( mkey ( #3:UName , s ) , #2:UName ; #3:UName ; #4:Nonce ;
   seskey ( #2:UName , #3:UName , n ( s , #0:Fresh ) ) ; #5:Key ) ) ,
- ( e ( mkey ( #2:UName , s ) , #2:UName ; #3:UName ; #4:Nonce ;
   seskey ( #2:UName , #3:UName , n ( s , #0:Fresh ) ) ; #5:Key ) ;
   e ( mkey ( #3:UName , s ) , #2:UName ; #3:UName ; #4:Nonce ;
   seskey ( #2:UName , #3:UName , n ( s , #0:Fresh ) ) ; #5:Key ) ) ,
+ ( ( #6:Msg ; e ( mkey ( b , s ) , a ; b ; #7:Nonce ; #8:Sessionkey ; #9:Key ) ) ;
    e ( #5:Key , #4:Nonce ; seskey ( #2:UName , #3:UName , n ( s , #0:Fresh ) ) ) ;
    n ( #3:UName , #10:Fresh ) ;
    e ( mkey ( #3:UName , s ) , #2:UName ; #3:UName ; #11:Sessionkey ;
    seskey ( #2:UName , #3:UName , n ( s , #0:Fresh ) ) ) ) ,
- ( ( #6:Msg ; e ( mkey ( b , s ) , a ; b ; #7:Nonce ; #8:Sessionkey ; #9:Key ) ) ;
    e ( #5:Key , #4:Nonce ; seskey ( #2:UName , #3:UName , n ( s , #0:Fresh ) ) ) ;
    n ( #3:UName , #10:Fresh ) ;
    e ( mkey ( #3:UName , s ) , #2:UName ; #3:UName ; #11:Sessionkey ;
    seskey ( #2:UName , #3:UName , n ( s , #0:Fresh ) ) ) ) ,
+ ( #6:Msg ; e ( mkey ( b , s ) , a ; b ; #7:Nonce ; #8:Sessionkey ; #9:Key ) ) ,
 - ( #6:Msg ; e ( mkey ( b , s ) , a ; b ; #7:Nonce ; #8:Sessionkey ; #9:Key ) ) ,
+ ( ( e ( #9:Key , #7:Nonce ; #8:Sessionkey ) ;
    e ( mkey ( b , s ) , a ; b ; #9:Key ; #8:Sessionkey ) ) ;
    e ( #9:Key , #7:Nonce ; #8:Sessionkey ) ; n ( b , #12:Fresh ) ;
```

```
      e ( mkey ( b , s ) , a ; b ; #9:Key ; #8:Sessionkey ) ) ,
- ( ( e ( #9:Key , #7:Nonce ; #8:Sessionkey ) ;
      e ( mkey ( b , s ) , a ; b ; #9:Key ; #8:Sessionkey ) ) ;
      e ( #9:Key , #7:Nonce ; #8:Sessionkey ) ; n ( b , #12:Fresh ) ;
      e ( mkey ( b , s ) , a ; b ; #9:Key ; #8:Sessionkey ) ) ,
+ ( e ( #9:Key , #7:Nonce ; #8:Sessionkey ) ;
      e ( mkey ( b , s ) , a ; b ; #9:Key ; #8:Sessionkey ) ) ,
- ( e ( #9:Key , #7:Nonce ; #8:Sessionkey ) ;
      e ( mkey ( b , s ) , a ; b ; #9:Key ; #8:Sessionkey ) )
```

La representación en notación informal sería la siguiente

(1) I(A)→S : G,Z,Ng
(2) S→I(A) : E(Kgs:G,Z,Ng,Kgz,Kt),E(Kzs:G,Z,Ng,Kgz,Kt)
(3) I(S)→B : E(Kgs:G,Z,Ng,Kgz,Kt), E(Kbs,a,b,Na,Kgz,Kt)
(4) B→I(A) : E(Kt:Ng,Kgz),(Kbs:a,b,Kt,Kgz),E(Kt:Ng,Kgz),Nb,E(Kbs:a,b,Kt,Kgz)
(5) I(A)→B : E(Kt:Ng,Kgz),E(Kbs:a,b,Kt:Kgz)

La visualización de los strands de la ejecución sería la siguiente:



7. Protocolos con propiedades algebraicas de primitivas criptográficas.

Las primitivas criptográficas tienen propiedades algebraicas que provienen del uso de funciones matemáticas, tales como la suma, multiplicación, or exclusiva o exponenciación modular. Aquí mostramos algunas de las propiedades algebraicas que se pueden utilizar en la ejecución de un protocolo o para atacarlo. También, las primitivas criptográficas utilizadas por los protocolos pueden presentar otras propiedades además de las descritas.

Or exclusiva.

El símbolo "<+>" representa al operador binario or exclusivo, también llamado XOR. Las propiedades de XOR son las siguientes:

- x <+> *(y <+> z)* = (x <+> y) <+> z (asociatividad).
- x <+> y = y <+> x (conmutatividad).
- x <+> 0 = x (elemento neutro).
- x <+> x = 0 (nilpotencia).

Esta operación es utilizada en muchos protocolos y despertó mucho interés durante los últimos años. H. Comon-Lundh and V. Shmatikov presentaron en [25] un procedimiento de decisión basado en técnicas de resolución para resolver el problema de seguridad de los protocolos criptográficos que utilizaban XOR. En [26], Y. Chevalier mejoró este resultado abstrayendo de las reglas del intruso utilizando reglas de llamada al oráculo, i.e, deduciendo reglas que satisfagan algunas condiciones. Como en una instancia del framework general, se obtiene que los problemas de seguridad para un amplio numero de sesiones son comunes a una amplia clase de protocolos donde el intruso puede explotar las propiedades del operador XOR.

En [27], H. Comon-Lundh y V. Cortier demostraron algunos resultados sobre la decidibilidad de una extensión de la lógica de primer orden de la clase de Skolem para la teoría ecuacional del operador XOR. Como una aplicación, ellos obtuvieron un resultado de decibilidad en el análisis formal en protocolos de seguridad ante la presencia de un numero de sesiones ilimitados. Ellos asumieron un numero finito de nonces, y supusieron que en cada transición, un agente puede copiar mas de un componente desconocido del mensaje recibido. Para otra subclase de lógica de primer orden, correspondiente a los autómatas de árboles de dos vías con XOR, K. Verma [28] también proporciona un resultado de decibilidad.

La herramienta Casper [29] considera la or exclusiva en el caso de la encriptación Vernam. La encriptación Vernam de m por m' es simplemente m <+> m'. G. Lowe lo modela añadiendo nuevas reglas de deducción al intruso. Por ejemplo, el intruso puede obtener m <+> m' de m y m'. Para obtener m' de m <+> m' y m, y obtener m <+> m' de m' <+> m.

Esta propiedad la trataremos en la sección 7.1.

Homomorfismo.

Consideraremos operadores que satisfacen las igualdades de la forma f(g(x,y))=g(f(x),f(y)). La propiedad del Homomorfismo se suele utilizar en algunos protocolos de votación. ElGamal [16], Paillier [21, 17], Goldwasser-Micali [18], Benaloh [14, 15], Naccache-Stern [19], Okamoto-Uchiyama [20] proporcionan primitivas criptográficas que verifican la propiedad del Homomorfismo. Mostramos el mecanismo ECB que induce la propiedad del homomorfismo {[x,y]}z=[{x}z,{y}z], que puede ser usada para atacar el protocolo de Needham-Schroeder-Lowe. Esta propiedad la trataremos en la Sección 7.2.

7.1. Needhamm-Schroeder-Lowe Mofified Protocol.

Este protocolo es una versión modificada del protocolo de autenticación de clave pública de Needham-Schroeder-Lowe. De hecho, el primer mensaje de la versión original es E(Kb,(Na,A)), con el nonce por primera vez en el campo de cifrado. Ahora se especificará como E(Kb,(A,Na)).

Esta diferencia aparentemente insignificante conduce a un ataque. En este caso, suponemos que cada agente conoce el principio de otras de clave pública.

Para este protocolo utilizaremos la or exclusiva junto a la cancelación de encriptación y desencriptación.

La descripción informal del protocolo proporcionada en la sección 3.1 de [2], pág. 15 es la siguiente:

(1) A→B : Kb:A,Na
(2) B→A : Ka:(Na,Nb),B
(3) A→B : Kb:Nb

Hemos utilizado para la implementación en Maude-NPA una revisión del protocolo usando XOR, que se hace en [5] , sería la siguiente:

```
(1) A→B : Kb:A,Na
(2) B→A : Ka:Nb,B<+>Na
(2) A→B : Kb:Nb
```

Los subtipos y operadores del protocolo, especificados en Maude-NPA quedarían de la siguiente manera:

```
--- Sort Information
  subsort Name Nonce Enc Key < NeNonceSet < Msg .
  subsort Name < Key .
  subsort Name < Public .
--- Encoding operators for public/private encryption
  op pk : Key Msg -> Enc [frozen] .
  op sk : Key Msg -> Enc [frozen] .

  --- Nonce operator
  op n : Name Fresh -> Nonce [frozen] .
  --- NeNonceSet
  op _<+>_ : NeNonceSet NeNonceSet -> NeNonceSet [frozen assoc comm] .

 --- Associativity operator
  op _;_ : Msg  Msg  -> Msg [assoc (e E) frozen] .
```

Las propiedades algebraicas serían las siguientes:

```
fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
  protecting PROTOCOL-EXAMPLE-SYMBOLS .

  ------------------------------------------------------------
  --- Overwrite this module with the algebraic properties
  --- of your protocol
  ------------------------------------------------------------

 vars XN YN : NNSet .

 eq d(K:Key, e(K:Key, Z:Msg)) = Z:Msg [metadata "variant"] .
 eq e(K:Key, d(K:Key, Z:Msg)) = Z:Msg [metadata "variant"] .

*** Exclusive or properties
eq XN <+> XN = null [metadata "variant"] .
eq XN <+> XN * YN = YN [metadata "variant"] .
eq XN <+> null = XN [metadata "variant"] .
endfm
```

En este protocolo existen dos roles: Alice y Bob.

La especificación del comportamiento de Alice es la siguiente:

```
    = :: r ::
--- Alice's Strand
     [ nil |  +(pk(B,A ; n(A,r))),
              -(pk(A, NB ; B <+> n(A,r))),
              +(pk(B, NB)), nil ]
```

La especificación del comportamiento de Bob es la siguiente:

```
    :: r ::
--- Bob's Strand
    [ nil |    -(pk(B,A ; NA)),
               +(pk(A, n(B,r) ; B <+> NA)),
               -(pk(B,n(B,r))), nil ]
```

En este protocolo vamos a considerar una configuración distinta que corresponde a la siguiente propiedad:

1. Una ejecución normal del protocolo (attack-state(0)).
2. Una ejecución donde el intruso averigua el nonce de (attack-state(1)).
3. Una ejecución donde Bob ha completado el protocolo creyendo que habla con Alice pero no es así (attack-state(2)).

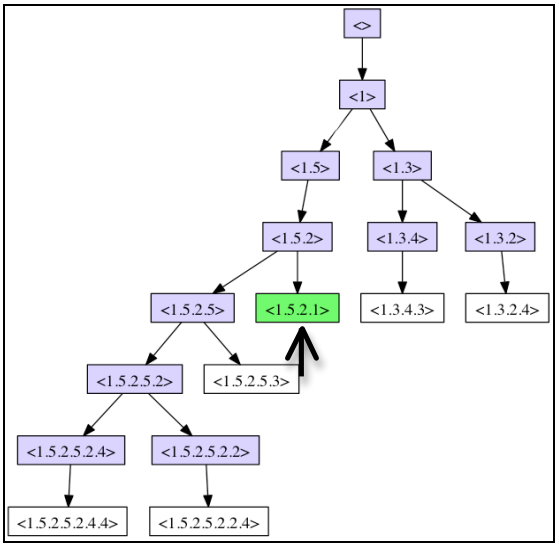
7.1.1. Attack-state(0). Ejecución normal.

Especificaríamos el attack-state(0) en Maude-NPA de la siguiente manera:

```
eq ATTACK-STATE(0)
  = :: r ::
    [ nil,  -(pk(b,a ; N)),
            +(pk(a, n(b,r) ; b <+> N)),
            -(pk(b,n(b,r)))  |  nil ]
    | | empty
    | | nil
    | | nil
    | | nil
  [nonexec] .
```

En este patrón de ataque se incluye sólo el participante del protocolo que responde. En el patrón se indica que el participante ha terminado su ejecución y el nonce recibido del iniciador del protocolo es desconocido y se representa con una variable N.

La visualización grafica de la ejecución del attack-state(0) sería el siguiente



Donde podemos ver que existe una solución, y por lo tanto una ejecución normal del protocolo.

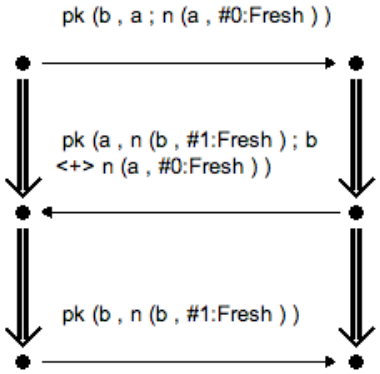
La secuencia de mensajes en la ejecución sería la siguiente

```
+ ( pk ( b , a ; n ( a , #0:Fresh ) ) ) ,
 - ( pk ( b , a ; n ( a , #0:Fresh ) ) ) ,
+ ( pk ( a , n ( b , #1:Fresh ) ; b <+> n ( a , #0:Fresh ) ) ) ,
 - ( pk ( a , n ( b , #1:Fresh ) ; b <+> n ( a , #0:Fresh ) ) ) ,
+ ( pk ( b , n ( b , #1:Fresh ) ) ) ,
 - ( pk ( b , n ( b , #1:Fresh ) ) )
```

El ataque también se expresaría de la siguiente manera:

(1) A→B : (Kb:(A,Na))
(2) B→A : (Ka:(Nb,B <+> Na))
(2) A→B : (Kb:Nb)

La visualización de los strands de la ejecución de la solución en el attack-state(0) sería la siguiente:



Donde podemos ver que se produce una ejecución normal.

El strand de Alice generado por la herramienta gráfica sería el siguiente:

```
:: #0:Fresh ::
 [ nil | + ( pk ( b , a ; n ( a , #0:Fresh ) ) ) ,
         - ( pk ( a , n ( b , #1:Fresh ) ; b <+> n ( a , #0:Fresh ) ) ) ,
         + ( pk ( b , n ( b , #1:Fresh ) ) ) , nil ] &
```

El strand de Bob generado por la herramienta gráfica sería el siguiente:

```
:: #1:Fresh ::
[ nil | - ( pk ( b , a ; n ( a , #0:Fresh ) ) ) ,
       + ( pk ( a , n ( b , #1:Fresh ) ; b <+> n ( a , #0:Fresh ) ) ) ,
        - ( pk ( b , n ( b , #1:Fresh ) ) ) , nil ]
```

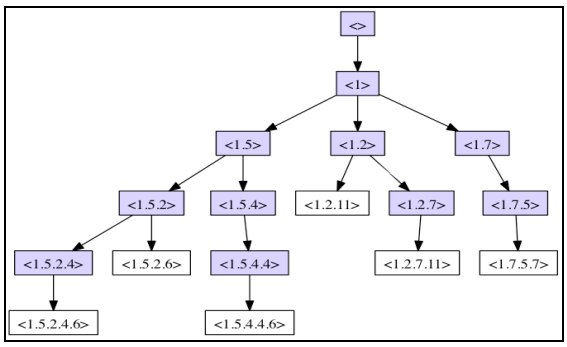
7.1.2. Attack-state(1). Comprobación de secreto.

El attack-state(1) especificado en Maude-NPA sería el siguiente:

```
eq ATTACK-STATE(1)
 = :: r ::
   [ nil, -(pk(b,a ; N)),
                          +(pk(a, (N <+> n(b,r)) ; b)),
                          -(pk(b,n(b,r)))  |  nil ]
   | | n(b,r) inI, empty
   | | nil
   | | nil
   | | nil
 [nonexec] .
```

En este caso indicamos que el intruso podría conocer la clave de sesión entre Alice y Bob generada por el Servidor.

El árbol de búsqueda que se obtiene usando la herramienta gráfica sería el siguiente.



Donde podemos ver que no existe una solución. Por lo tanto es seguro ante un ataque de obtención de la clave de sesión.

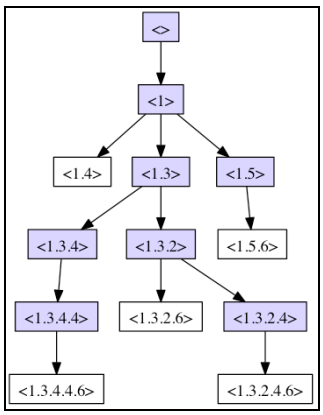
7.1.3. Attack-state(2). Comprobación de autenticación.

El attack-state(2) especificado en Maude-NPA sería el siguiente:

```
eq ATTACK-STATE(2)
= :: r ::
  [ nil, -(pk(b,a ; NA)),
        +(pk(a, n(b,r) ; b <+> NA)),
        -(pk(b,n(b,r)))  |  nil ]
  | |  empty
  | | nil
  | | nil
  | | never *** for authentication
  (:: r' ::
  [ nil, +(pk(b,a ; NA)),
         -(pk(a, n(b,r) ; b <+> NA)),
         +(pk(b, n(b,r)))  |  nil ]
  & S:StrandSet
  | | K:IntruderKnowledge)
[nonexec] .
```

En este caso usaríamos el strand de Alice como patrón de autenticación.

El árbol de búsqueda que se obtiene al ejecutar la herramienta gráfica sería el siguiente:



Donde podemos ver que no existe una solución. Por lo tanto el protocolo es seguro ante un ataque de autenticación.

7.2. Needham-Schroeder-Lowe Protocol With ECB

Este protocolo consiste en dividir el mensaje en bloques de n bits y encriptación de cada uno de ellos por separado. El protocolo de Needham-Schroeder-Lowe, que se utiliza para la autenticación mutua de los dos participantes principales, es defectuoso si el cifrado se realiza utilizando el BCE (Electronic Code Book (ECB) Libro de Código Electrónico (BCE)). Al final del protocolo, cada uno de los participantes principales está convencido de que habla con el participante de la derecha, y de compartir los secretos de Na y Nb. Una propiedad básica de BCE es que {A, B, C} Pk = {A} Pk, {B} Pk, {C} Pk, donde el tamaño de los bloques A, B, C es un múltiplo del bloque longitud usado por el algoritmo de cifrado.

La propiedad algebraica que se utiliza en este protocolo es la del homomorfismo. Esta propiedad se especificaría de la siguiente manera: {[x,y]}z=[{x}z,{y}z]. Esta propiedad se utilizará para atacar el protocolo de Needham-Schroeder-Lowe.

La descripción informal del protocolo proporcionada en la sección 3.5 de [2], pág. 21 es la siguiente:

(1) A→B: Kb:(A,Na)
(2) B→A: Ka:(B,Nb,Na)
(3) A→B: Kb:Nb

Los subtipos y operadores del protocolo, especificados en Maude-NPA, quedarían de la siguiente manera:

```
 --- Sort Information
  subsort Name Nonce Enc Key < Msg .
  subsort Name < Key .
  subsort Name < Public .

--- Encoding operators for public/private encryption
  op pk : Key Msg -> Enc [frozen] .
  op sk : Key Msg -> Enc [frozen] .
--- Nonce operator
  op n : Name Fresh -> Nonce [frozen] .

  --- Principals
  op a : -> Name . --- Alice
  op b : -> Name . --- Bob
  op i : -> Name . --- Intruder
--- Associativity operator
  op _;_ : Msg  Msg  -> Msg [gather (e E) frozen] .
```

Las propiedades algebraicas serían las siguientes:

```
fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
  protecting PROTOCOL-EXAMPLE-SYMBOLS .

  ------------------------------------------------------------
  --- Overwrite this module with the algebraic properties
  --- of your protocol
  ------------------------------------------------------------

eq pk(X:Msg ; Y:Msg, K:Key) = pk(X:Msg, K:Key) ; pk(Y:Msg, K:Key)
[nonexec label homomorphism metadata "builtin-unify"] .

endfm
```

En este protocolo existen dos roles: Alice y Bob.

La especificación del comportamiento de Alice es la siguiente:

```
 = :: r ::
  ---Alice's Strand
    [ nil |  +(pk(B, A ; n(A,r))),
             -(pk(A, B ; N ; n(A,r))),
             +(pk(B, N)), nil ] &
```

La especificación del comportamiento de Bob es la siguiente:

```
   :: r ::
    --- Bob's Strand
    [ nil |  -(pk(B,A ; N)),
           +(pk(A, B ; n(B,r) ; N)),
            -(pk(B,n(B,r))), nil ]
```

En este protocolo vamos a considerar dos configuraciones que corresponden a las siguientes propiedades:

1. Una ejecución normal del protocolo (attack-state(0)).
2. Una ejecución donde el intruso averigua "{Na, Nb} PK(A)" (attack-state(1)). Este conocimiento es necesario para el ataque documentado en [2].
3. Una ejecución donde Bob ha completado el protocolo creyendo que habla con Alice pero no es así (attack-state(2)).

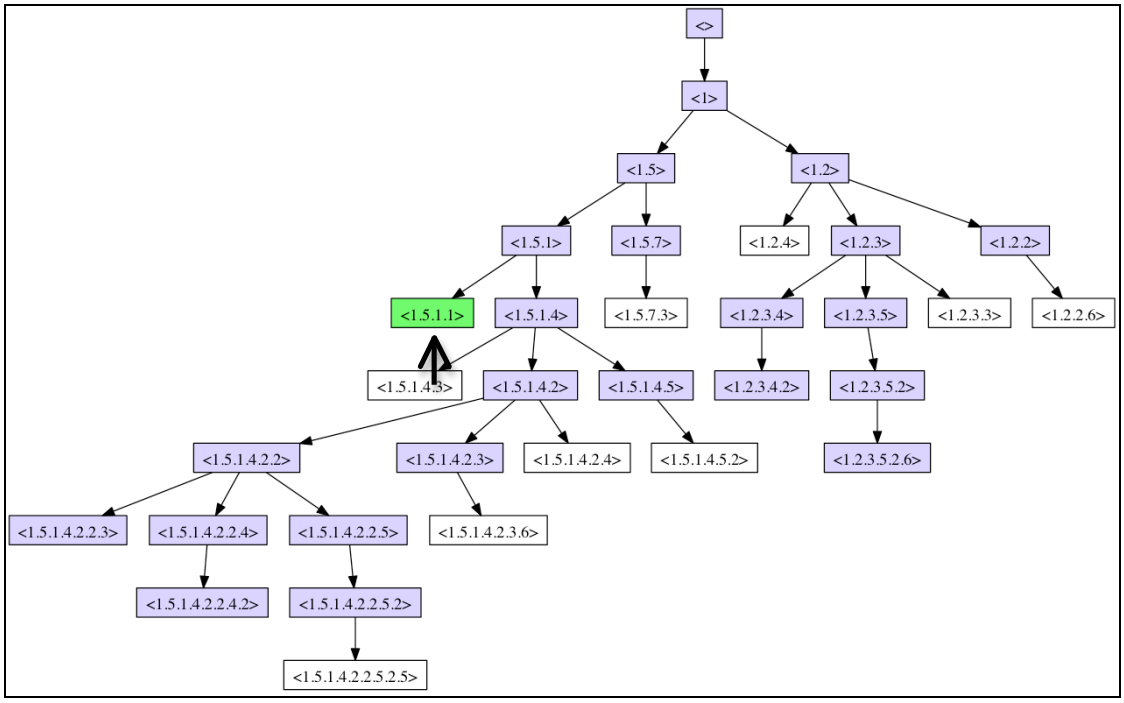
7.2.1. Attack-state(0). Ejecución normal.

Especificaríamos el attack-state(0) en Maude-NPA de la siguiente manera:

```
eq ATTACK-STATE(0)
  = :: r ::
    [ nil, -(pk(b,a ; N)),
          +(pk(a, b; n(b,r) ; N)),
           -(pk(b,n(b,r)))  |  nil ]
    | | empty
    | | nil
    | | nil
   [nonexec] .
```

En este patrón de ataque se incluye sólo el participante del protocolo que responde. En el patrón se indica que el participante ha terminado su ejecución y el nonce recibido del iniciador del protocolo es desconocido y se representa con una variable N.

Mostramos el árbol de búsqueda a continuación:



Donde podemos ver que existe una solución, y por lo tanto una ejecución normal del protocolo.

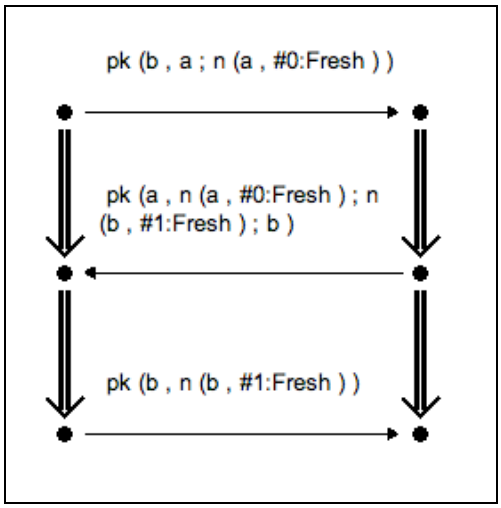
La secuencia de mensajes en la ejecución sería la siguiente:

```
+ ( pk ( b , a ; n ( a , #0:Fresh ) ) ) ,
 - ( pk ( b , a ; n ( a , #0:Fresh ) ) ) ,
+ ( pk ( a , n ( a , #0:Fresh ) ; n ( b , #1:Fresh ) ; b ) ) ,
 - ( pk ( a , n ( a , #0:Fresh ) ; n ( b , #1:Fresh ) ; b ) ) ,
+ ( pk ( b , n ( b , #1:Fresh ) ) ) ,
 - ( pk ( b , n ( b , #1:Fresh ) ) )
```

El ataque también se expresaría de la siguiente manera:

```
(1) A→B : Kb:(A,Na)
(2) B→A : Ka:(B,Nb,Na)
(3) A→B : Kb:Nb
```

La visualización de los strands de la ejecución de la solución en el attack-state(0) sería la siguiente:



Donde podemos ver que se produce una ejecución normal.

El strand de Alice generado por la herramienta gráfica sería el siguiente:

```
:: r: Fresh ::
[ nil | + ( pk ( b , a ; n ( a , r:Fresh ) ) ) ,
        - ( pk ( a , b ; n ( b , r:Fresh ) ; n ( a , r:Fresh ) ) ) ,
        + ( pk ( b , n ( b , r:Fresh ) ) ) , nil ] &
```

El strand de Bob generado por la herramienta gráfica sería el siguiente:

```
:: r: Fresh ::
[ nil | - ( pk ( b , a ; n ( a , r:Fresh ) ) ) ,
       + ( pk ( a , b ; n ( b , r:Fresh ) ; n ( a , r:Fresh ) ) ) ,
        - ( pk ( b , n ( b , r:Fresh ) ) ) , nil ]
```

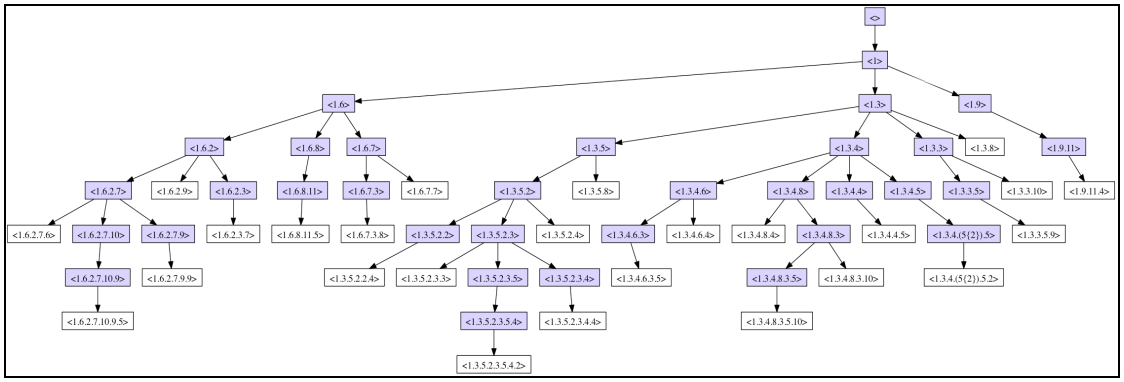
7.2.2. Attack-state(1). Comprobación de secreto.

Especificaríamos el attack-state(1) en Maude-NPA de la siguiente manera:

```
eq ATTACK-STATE(1)
= :: r ::
 [ nil, -(pk(b,a ; N)),
       +(pk(a, b ; n(b,r) ; N)),
        -(pk(b,n(b,r))) | nil ]
 | | pk(a, N ; n(b,r)) inI
 | | nil
 | | nil
 | | nil
[nonexec] .
```

En este caso indicamos que el intruso podría conocer "pk(a, N ; n(b,r))". En el ataque documentado en [2] el intruso puede extraer {Na, Nb}PK(A) de {Na,Nb,B}PK(A). Con lo cual, indicamos que el intruso ya conoce "pk(a, N ; n(b,r))" para dar pie a este ataque.

El árbol de búsqueda que se obtiene al ejecutar la herramienta gráfica sería el siguiente:



Donde podemos ver que no existe una solución. Por lo tanto el protocolo es seguro ante este ataque. Indicar que este ataque sólo es posible si la talla de cada nonce y la variable de cada protagonista del protocolo son múltiplos del tamaño máximo de la cifra impuesta por la función de encriptación.

7.2.3. Attack-state(2). Comprobación de autenticación.

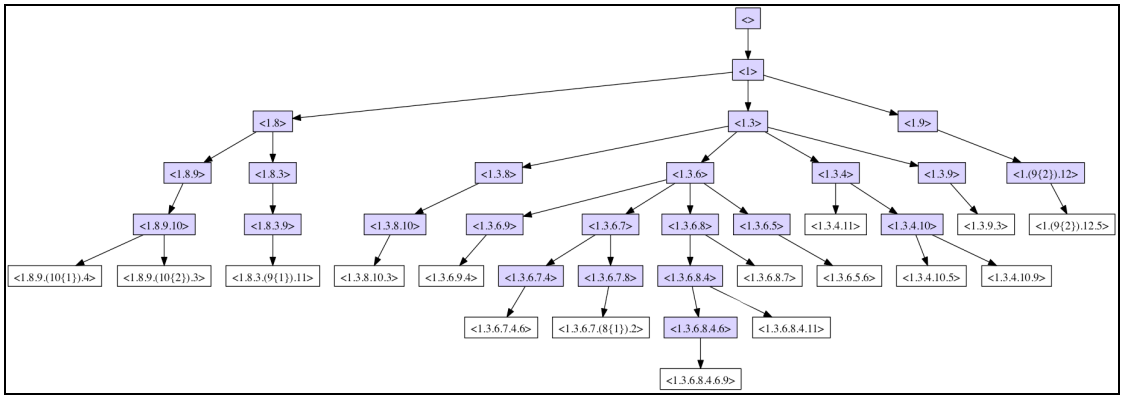
Especificaríamos el attack-state(2) en Maude-NPA de la siguiente manera:

```
eq ATTACK-STATE(2)
= :: r ::
 [ nil, -(pk(b,a ; N)),
       +(pk(a, b ; n(b,r) ; N)),
        -(pk(b,n(b,r))) | nil ]
 | | pk(a, b ; n(b,r) ; N) inI
 | | nil
 | | nil
 | | never
 *** Pattern for authentication
 (:: R:FreshSet ::
 [ nil | +(pk(b,a ; N)),
         -(pk(a, b ; n(b,r) ; N)),
        +(pk(b,n(b,r))),  nil ]
  & S:StrandSet | | K:IntruderKnowledge)
```

[**nonexec**] .

En este caso usaríamos el strand de Alice como patrón de autenticación.

El árbol de búsqueda que se obtiene al ejecutar la herramienta gráfica sería el siguiente:



Donde podemos ver que no existe una solución. Por lo tanto el protocolo es seguro ante un ataque de autenticación.

8. Referencias.

[1] John Clark and Jeremy Jacob: "A Survey of Authentication Protocol".

[2] Véronique Cortier , Stéphanie Delaune, and Pascal Lafourcade: "A Survey of Algebraic Properties Used in Cryptographic Protocols".

[3] Cryptographic Protocol Type Checker: http://cryptyc.cs.depaul.edu/OLDiso5.html

[4] "A Calculus for Cryptographic Protocols. The Spi Calculus" : Martín Abadi and Andrew D. Gordon

[5] "Deduction with XOR Constraints in Security API Modelling" : Graham Steel

[6] S. Escobar, C. Meadows, and J. Meseguer. Maude-NPA: Cryptographic Protocol Analysis Modulo Equational Properties. In A. Aldini, G. Barthe, and R. Gorrieri, editors, FOSAD 2008/2009 Tutorial Lectures, Lecture Notes in Computer Science. Springer, 2009. to appear.

[7] Manuel Clavel, Francisco Dur´an, Steven Eker, Patrick Lincoln, Narciso Mart´ı-Oliet, Jos´e Meseguer, and Carolyn L. Talcott, editors. All About Maude -A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, volume 4350 of Lecture Notes in Computer Science. Springer, 2007.

[8] F. J. Thayer Fabrega, J. Herzog, and J. Guttman. Strand Spaces: What Makes a Security Protocol Correct? Journal of Computer Security, 7:191– 230, 1999.

[9] Manuel Clavel, Francisco Dur´an, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Mart´ı-Oliet, Jos´e Meseguer, and Carolyn Talcott. Unification and Narrowing in Maude 2.4. In Ralf Treinen, editor, Proceedings of the 20th International Conference on Rewriting Techniques and Applications (RTA'09), Lecture Notes in Computer Science. Springer, June-July 2009. To appear.

[10] D. Dolev and A. Yao. On the security of public key protocols. Transaction on Information Theory, 29(2):198–208, 1983.

[11] Santiago Escobar, Catherine Meadows, and Jose Meseguer. Maude-NPA, Version 1.0, March 2009. Available at http://maude.cs.uiuc.edu/ tools/Maude-NPA.

[12] Roger M. Needham and Michael D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. Commun. ACM, 21(12):993– 999, 1978.

[13] F. Javier Thayer, Jonathan C. Herzog, and Joshua D. Guttman. Strand Spaces: Proving Security Protocols Correct. Journal of Computer Security, 7(1), 1999.

[14] J. Benaloh. Secret sharing homomorphisms: Keeping shares of a secret sharing. In *Proc. Advances in Cryptology (CRYPTO'86)*, vol. 263 of *LNCS*, p. 251–260, Santa Barbara (California, USA), 1987.

Springer-Verlag.

[15] J. Cohen and M. Fischer. A robust and verifiable cryptographically secure election scheme. In *Proc. 26th Annual Symposium on Foundations of Computer Science (FOCS'85)*, p. 372–382, Portland (Oregon, USA), 1985. IEEE Comp.Soc. Press.

[16] T. El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proc. Advances in Cryptology (CRYPTO'84)*, vol. 196 of *LNCS*, p. 10–18, Santa Barbara (California, USA), 1985. Springer-Verlag.

[17] P.-A. Fouque, G. Poupard, and J. Stern. Sharing decryption in the context of voting or lotteries. In *Proc. 4th International Conference on Financial Cryptography (FC'00)*, vol. 1962 of *LNCS*, p. 90–104, Anguilla (BritishWest Indies), 2001. Springer-Verlag.

[18] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[19] D. Naccache and J. Stern. A new public-key cryptosystem. *Proc. International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT'97)*, 1233:27–37, 1997.

[20] Okamoto and S. Uchiyama. A new public-key cryptosystemas secure as factoring. In *Proc. International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT'98)*, vol. 1403, p. 308–318, Helsinki (Finland), 1998. Springer-Verlag. LNCS.

[21] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proc. International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT'99)*, vol. 1592 of *LNCS*, p. 223–238,Prague (Czech Republic), 1999. Springer-Verlag.

[22] The Maude System webpage : http://maude.cs.uiuc.edu/ y http://maude.cs.uiuc.edu/tools/Maude-NPA/

[23] Maude-NPA GUI : http://users.dsic.upv.es/~ssantiago/Maude-NPA_GUI/

[24] Michael *Burrows* Martiin *Abadi* Roger *Needham*: "*A Logic of Authentication*".

[25] H. Comon-Lundh and V. Shmatikov. Intruder deductions, constraint solving and insecurity decision in presence of exclusive or. In *Proc. of 18th Annual IEEE Symposium on Logic in Computer Science (LICS'03),* p. 271–280, Ottawa (Canada), 2003. IEEE Comp. Soc. Press.

[26] Y. Chevalier, R. Küsters, M. Rusinowitch, and M. Turuani. An NP decision procedure for protocol insecurity with XOR. In *Proc. of 18th Annual IEEE Symposium on Logic in Computer Science (LICS'03),* p. 261–270, Ottawa (Canada), 2003. IEEE Comp. Soc. Press.

[27] H. Comon-Lundh and V. Cortier. New decidability results for fragments of first-order logic and application to cryp-

tographic protocols. In *Proc. 14th International Conference on Rewriting Techniques and Applications (RTA'2003),* vol. 2706 of *LNCS,* p. 148–164, Valencia (Spain), 2003. Springer-Verlag.

[28] K. N. Verma. Two-way equational tree automata for AC-like theories: Decidability and closure properties. In *Proc. 14th International Conference on Rewriting Techniques and Applications (RTA'03),* vol. 2706 of *LNCS,* p. 180–196, Valencia (Spain), 2003. Springer-Verlag.

[29] G. Lowe. Casper: A compiler for the analysis of security protocols. In *Proc. 10th Computer Security Foundations Workshop (CSFW'97),* p. 18–30, Rockport (Massachusetts, USA), 1997. IEEE Comp. Soc. Press.